

IBM WebSphere InterChange Server



# Map Development Guide

*Version 4.3.0*



IBM WebSphere InterChange Server



# Map Development Guide

*Version 4.3.0*

**Note!**

Before using this information and the product it supports, read the information in "Notices" on page 509.

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 to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, e-mail [doc-comments@us.ibm.com](mailto: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 1997, 2004. All rights reserved. US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

© **Copyright International Business Machines Corporation 1997, 2004. All rights reserved.**

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

---

# Contents

<b>About this document</b> . . . . .	<b>xi</b>
Audience . . . . .	xi
How to use this manual . . . . .	xi
Related documents . . . . .	xii
Typographic conventions . . . . .	xii
<b>New in this release</b> . . . . .	<b>xiii</b>
New in release 4.3 . . . . .	xiii
New in release 4.2.2 . . . . .	xiii
New in release 4.2.1 . . . . .	xiii
New in release 4.2.0 . . . . .	xiv
New in release 4.1.1 . . . . .	xiv
New in release 4.1.0 . . . . .	xv
New in release 4.0.1 . . . . .	xv
New in release 4.0.0 . . . . .	xv
<b>Part 1. Maps</b> . . . . .	<b>1</b>
<b>Chapter 1. Introduction to map development</b> . . . . .	<b>3</b>
About data mapping . . . . .	3
Maps: A closer look . . . . .	5
Tools for map development . . . . .	7
Overview of map development . . . . .	11
<b>Chapter 2. Creating maps</b> . . . . .	<b>15</b>
Overview of Map Designer . . . . .	15
Creating a map: Basic steps . . . . .	30
Mapping standards . . . . .	52
<b>Chapter 3. Working with maps</b> . . . . .	<b>55</b>
Opening and closing a map . . . . .	55
Specifying map property information . . . . .	58
Designing maps for bidirectional languages . . . . .	60
Using map documents . . . . .	60
Using map automation . . . . .	65
Finding information in a map . . . . .	72
Finding and replacing text . . . . .	73
Printing a map . . . . .	74
Deleting objects . . . . .	74
Using execution order . . . . .	77
Creating polymorphic maps . . . . .	78
Importing and exporting maps from InterChange Server . . . . .	79
Converting old maps . . . . .	80
<b>Chapter 4. Compiling and testing maps</b> . . . . .	<b>83</b>
Checking the transformation code . . . . .	83
Validating a map . . . . .	84
Compiling a map . . . . .	84
Compiling a set of maps . . . . .	86
Testing maps . . . . .	87
Doing advanced debugging . . . . .	94
Testing maps that contain relationships . . . . .	95
Debugging maps . . . . .	100

<b>Chapter 5. Customizing a map</b>	<b>103</b>
Overview of Activity Editor	103
Working with activity definitions	112
Exporting Web services into Activity Editor	159
Using bidirectional functionality in Activity Editor	162
Importing Java packages and other custom code	164
Using variables	169
More attribute transformation methods	173
Reusing map instances	185
Handling exceptions	186
Creating custom data validation levels	187
Understanding map execution contexts	189
Mapping child business objects	193
More on using submaps	198
Executing database queries	203

---

## **Part 2. Relationships** **221**

### **Chapter 6. Introduction to relationships** **223**

What is a relationship?	223
Relationships: A closer look	229
Overview of the relationship development process	235

### **Chapter 7. Creating relationship definitions** **237**

Overview of Relationship Designer	237
Creating a relationship definition	243
Defining identity relationships	244
Defining lookup relationships	246
Creating the relationship table schema	248
Copying relationship and participant definitions	248
Renaming relationship or participant definitions	249
Specifying advanced relationship settings	249
Deleting a relationship definition	253
Optimizing a relationship	253

### **Chapter 8. Implementing relationships** **257**

Implementing a relationship	257
Using lookup relationships	258
Using simple identity relationships	263
Using composite identity relationships	274
Managing child instances	282
Setting the verb	285
Performing foreign key lookups	290
Maintaining custom relationships	295
Writing safe relationship code	297
Executing queries in the relationship database	299
Loading and unloading relationships	309

---

## **Part 3. Mapping API Reference** **313**

### **Chapter 9. BaseDLM class** **315**

getConnection()	315
getName()	317
getRelConnection()	318
implicitDBTransactionBracketing()	319
isTraceEnabled()	319
logError(), logInfo(), logWarning()	320
raiseException()	321
releaseRelConnection()	323

trace()	324
<b>Chapter 10. BusObj class</b>	<b>327</b>
Exceptions and exception types	328
Syntax for traversing hierarchical business objects	328
copy()	329
duplicate()	330
equalKeys()	330
equals()	331
equalsShallow()	332
exists()	332
getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(),	
getString()	333
getLocale()	335
getType()	335
getVerb()	336
isBlank()	336
isKey()	336
isNull()	337
isRequired()	338
keysToString()	338
set()	339
setContent()	340
setDefaultAttrValues()	341
setKeys()	341
setLocale()	342
setVerb()	342
setVerbWithCreate()	343
setWithCreate()	343
toString()	344
validData()	345
Deprecated methods	345
<b>Chapter 11. BusObjArray class</b>	<b>347</b>
addElement()	348
duplicate()	348
elementAt()	349
equals()	349
getElements()	350
getLastIndex()	350
max()	350
maxBusObjArray()	351
maxBusObjs()	352
min()	353
minBusObjArray()	354
minBusObjs()	355
removeAllElements()	356
removeElement()	356
removeElementAt()	357
setElementAt()	357
size()	358
sum()	358
swap()	358
toString()	359
<b>Chapter 12. CwBidiEngine class</b>	<b>361</b>
BiDiBOTransformation()	361
BiDiBusObjTransformation()	362
BiDiStringTransformation()	363

<b>Chapter 13. CwDBConnection class . . . . .</b>	<b>365</b>
beginTransaction() . . . . .	365
commit() . . . . .	366
executePreparedSQL() . . . . .	367
executeSQL() . . . . .	368
executeStoredProcedure() . . . . .	370
getUpdateCount() . . . . .	371
hasMoreRows() . . . . .	371
inTransaction() . . . . .	372
isActive() . . . . .	372
nextRow() . . . . .	373
release() . . . . .	373
rollBack() . . . . .	374
<b>Chapter 14. CwDBStoredProcedureParam class . . . . .</b>	<b>377</b>
CwDBStoredProcedureParam() . . . . .	377
getParamType() . . . . .	378
getValue() . . . . .	379
<b>Chapter 15. DtpConnection class . . . . .</b>	<b>381</b>
beginTran() . . . . .	381
commit() . . . . .	382
executeSQL() . . . . .	383
execStoredProcedure() . . . . .	384
getUpdateCount() . . . . .	385
hasMoreRows() . . . . .	385
inTransaction() . . . . .	386
nextRow() . . . . .	386
rollBack() . . . . .	387
<b>Chapter 16. DtpDataConversion class . . . . .</b>	<b>389</b>
getType() . . . . .	389
isOKToConvert() . . . . .	390
toBoolean() . . . . .	392
toDouble() . . . . .	393
toFloat() . . . . .	393
toInteger() . . . . .	394
toPrimitiveBoolean() . . . . .	395
toPrimitiveDouble() . . . . .	395
toPrimitiveFloat() . . . . .	396
toPrimitiveInt() . . . . .	397
toString() . . . . .	398
<b>Chapter 17. DtpDate class . . . . .</b>	<b>399</b>
DtpDate() . . . . .	401
addDays() . . . . .	402
addWeekdays() . . . . .	403
addYears() . . . . .	404
after() . . . . .	405
before() . . . . .	406
calcDays() . . . . .	406
calcWeekdays() . . . . .	407
get12MonthNames() . . . . .	408
get12ShortMonthNames() . . . . .	408
get7DayNames() . . . . .	408
getCWDate() . . . . .	409
getDayOfMonth() . . . . .	409
getDayOfWeek() . . . . .	410
getHours() . . . . .	410
getIntDay() . . . . .	410



getIntDayOfWeek()	411
getIntMilliseconds()	411
getIntMinutes()	411
getIntMonth()	412
getIntSeconds()	412
getIntYear()	412
getMSSince1970()	413
getMaxDate()	413
getMaxDateBO()	414
getMinDate()	415
getMinDateBO()	417
getMinutes()	418
getMonth()	418
getNumericMonth()	418
getSeconds()	419
getShortMonth()	419
getYear()	420
set12MonthNames()	420
set12MonthNamesToDefault()	421
set12ShortMonthNames()	421
set12ShortMonthNamesToDefault()	421
set7DayNames()	422
set7DayNamesToDefault()	422
toString()	422
<b>Chapter 18. DtpMapService class</b>	<b>425</b>
runMap()	425
<b>Chapter 19. DtpSplitString class</b>	<b>427</b>
DtpSplitString()	427
elementAt()	428
firstElement()	428
getElementCount()	429
getEnumeration()	430
lastElement()	430
nextElement()	430
prevElement()	431
reset()	432
<b>Chapter 20. DtpUtils class</b>	<b>433</b>
padLeft()	433
padRight()	433
stringReplace()	434
truncate()	435
<b>Chapter 21. IdentityRelationship class</b>	<b>437</b>
addMyChildren()	437
deleteMyChildren()	439
foreignKeyLookup()	440
foreignKeyXref()	442
maintainChildVerb()	444
maintainCompositeRelationship()	446
maintainSimpleIdentityRelationship()	448
updateMyChildren()	450
<b>Chapter 22. CxExecutionContext class</b>	<b>453</b>
Static constants	453
CxExecutionContext()	453
getContext()	454
setContext()	454

<b>Chapter 23. MapExeContext class</b> . . . . .	<b>457</b>
getConnectionName() . . . . .	457
getInitiator() . . . . .	457
getLocale() . . . . .	458
getOriginalRequestBO() . . . . .	459
setConnName() . . . . .	460
setInitiator() . . . . .	460
setLocale() . . . . .	461
Deprecated methods . . . . .	462
<b>Chapter 24. Participant class.</b> . . . . .	<b>463</b>
Participant() . . . . .	463
getBusObj(), getString(), getLong(), getInt(), getDouble(),	
getFloat(), getBoolean() . . . . .	465
getInstanceId() . . . . .	465
getParticipantDefinition() . . . . .	466
getRelationshipDefinition() . . . . .	466
set() . . . . .	467
setInstanceId() . . . . .	467
setParticipantDefinition() . . . . .	468
setRelationshipDefinition() . . . . .	468
<b>Chapter 25. Relationship class</b> . . . . .	<b>471</b>
addParticipant() . . . . .	472
create() . . . . .	474
deactivateParticipant() . . . . .	475
deactivateParticipantByInstance() . . . . .	476
deleteParticipant() . . . . .	477
deleteParticipantByInstance() . . . . .	478
getNewID() . . . . .	479
retrieveInstances() . . . . .	480
retrieveParticipants() . . . . .	482
updateParticipant() . . . . .	483
updateParticipantByInstance() . . . . .	484
Deprecated methods . . . . .	485
<b>Chapter 26. UserStoredProcedureParam class</b> . . . . .	<b>487</b>
UserStoredProcedureParam() . . . . .	487
getParamDataJavaObj() . . . . .	488
getParamDataJDBC() . . . . .	489
getParamIndex() . . . . .	489
getParamIOType() . . . . .	490
getParamName() . . . . .	491
getParamValue() . . . . .	491
setParamDataJavaObj() . . . . .	492
setParamDataJDBC() . . . . .	492
setParamIndex() . . . . .	493
setParamIOType() . . . . .	493
setParamName() . . . . .	494
setParamValue() . . . . .	494
<b>Part 4. Appendixes</b> . . . . .	<b>495</b>
<b>Appendix A. Message files.</b> . . . . .	<b>497</b>
Message location . . . . .	497
Format for map messages . . . . .	500
Maintaining the files . . . . .	502
Operations that use message files. . . . .	502

<b>Appendix B. Attribute properties</b> . . . . .	<b>507</b>
<b>Notices</b> . . . . .	<b>509</b>
Programming interface information . . . . .	510
Trademarks and service marks. . . . .	511
<b>Index</b> . . . . .	<b>513</b>



---

## About this document

IBM<sup>(R)</sup> WebSphere<sup>(R)</sup> 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 provides an introduction to the use of maps and relationships and describes how to use IBM WebSphere tools for creating and modifying them.

---

## Audience

This document is for connector developers, collaboration developers, and IBM WebSphere consultants who create or modify business object definitions or maps.

---

## How to use this manual

This manual is organized as follows.

---

### Part I: Maps

- |  |  |
|--|--|
| Chapter 1, "Introduction to map development" | Is an overview of maps and the WebSphere Business Integration mapping tools.                                       |
| Chapter 2, "Creating maps"                   | Provides an introduction to the use of Map Designer for the creation and modification of maps.                     |
| Chapter 3, "Working with maps"               | Describes some advanced features of Map Designer that you might use after creating maps.                           |
| Chapter 4, "Compiling and testing maps"      | Describes how to compile a map into its executable form and how to run a test run to verify the map's correctness. |
| Chapter 5, "Customizing a map"               | Describes how to implement maps.   |

### Part II: Relationships

- |  |  |
|--|--|
| Chapter 6, "Introduction to relationships"     | Provides an introduction to relationships, including the kinds of relationships that WebSphere Business Integration supports and the way the system implements a relationship. |
| Chapter 7, "Creating relationship definitions" | Provides an introduction to the use of Relationship Designer for the creation and modification of relationship definitions.  |
| Chapter 8, "Implementing relationships"        | Describes how to implement relationships.  |

### Part III: Mapping API Reference

---

---

Chapter 9, "BaseDLM class"	Contain reference pages for methods of classes in the Mapping API.
Chapter 10, "BusObj class"	
Chapter 11, "BusObjArray class"	
Chapter 12, "CwBidiEngine class," on page 361	
Chapter 13, "CwDBConnection class"	
Chapter 14, "CwDBStoredProcedureParam class"	
Chapter 15, "DtpConnection class"	
Chapter 16, "DtpDataConversion class"	
Chapter 18, "DtpMapService class"	
Chapter 19, "DtpSplitString class"	
Chapter 20, "DtpUtils class"	
Chapter 21, "IdentityRelationship class"	
Chapter 22, "CxExecutionContext class," on page 453	
Chapter 24, "Participant class"	
Chapter 23, "MapExeContext class"	
Chapter 25, "Relationship class"	
Chapter 26, "UserStoredProcedureParam class"	

Appendix A, "Message files"

Appendix B, "Attribute properties"

---

## Related documents

The complete set of documentation available with this product describes the features and components common to all Websphere InterChange Server installations, and includes reference material on specific components.

You can install the documentation or read it directly online at the IBM WebSphere InterChange Server InfoCenter, located at:

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

This site contains simple instructions for downloading, installing, and viewing the documentation.

Before using this document, you should read the *Technical Introduction to IBM WebSphere InterChange Server* to understand how collaborations and connectors use business objects and maps.

---

## Typographic conventions

This document uses the following conventions:

---

<code>courier font</code>	Indicates a literal value, such as a command name, information that you type, or information that the system prints on the screen.
<i>italic or italic</i>	Indicates a variable name, title name, or new term the first time that it appears
<i>blue outline</i>	A blue outline, which is visible only when you view the manual online, indicates a cross-reference hyperlink. Click inside the outline to jump to the object of the reference.
<i>ProductDir</i>	Represents the directory where the product is installed.

---

---

## New in this release

This section describes the new and changed features of IBM WebSphere InterChange Server 4.3 and its associated tools for map and relationship development, which are covered in this document.

---

### New in release 4.3

- Support is provided for the Map Automation tool, which allows you to create maps automatically between business objects with similar attributes and to generate reverse maps for any given maps.
- Activity Editor has been enhanced for Java source parsing and for importing and exporting activity settings configurations.
- The Activity Editor group block has been enhanced with an icon representation to manage components better when the activity group is reused.
- Support is provided for configuring standard function blocks in the Preference dialog for direct use in Map Designer.
- Support is provided for Web services enablement.
- Support is provided for bidirectional language capability for naming maps.
- Additional support is provided for handling primitives in the DtpDataConversion class.
- A new chapter is provided for the CwBidiEngine class.
- A new chapter is provided for the CxExecutionContext class.

---

### New in release 4.2.2

- A new "My Library" feature has been added to the Activity Editor with the capability to add a custom library.
- Many new function blocks have been added to the Activity Editor, including function blocks for implementing relationships.
- Some dialogs, icons, and other interface design components have been modified in Map Designer and Relationship Designer.
- The Activity Editor has changes to its interface due to the addition of new function blocks and "My Library" feature.
- Support is provided for modifying the Validation preference in the Activity Editor for validating the sameness of parameter types for linking.
- Support is provided for modifying the Compiler Classpath preference for compiling maps and collaboration templates from System Manager.

---

### New in release 4.2.1

- Support is provided for finding and replacing text.
- New function blocks have been added for logging and tracing, including message parameters. This information is in Chapter 3, "Working with maps," on page 55..
- Support is provided for resizing editing components in the Activity Editor.
- Support is provided on using the Static Lookup relationship function block in the Activity Editor.

- Support is provided for changing selected map properties from the server component management view.
- Support is provided for changing the cached and reload relationship properties from the server component management view.
- The actions for the SERVICE\_CALL\_RESPONSE calling context have been updated for the maintainSimpleIdentityRelationship() method in Chapter 8, "Implementing relationships," on page 257..
- An example has been provided for the getInitiator() method in Chapter 23, "MapExeContext class," on page 457..

---

## New in release 4.2.0

- The CrossWorlds name is no longer used to describe an entire system or to modify the names of components or tools. For example, "CrossWorlds System Manager" is now "System Manager," and "CrossWorlds Interchange Server" is now "WebSphere Interchange Server." Name changes have been implemented, accordingly, in this document.
- Map Designer has changes to its interface, including new dialogs, menu, options and toolbar buttons. These changes are described in Chapter 2, "Creating maps," on page 15, Chapter 3, "Working with maps," on page 55 and Chapter 4, "Compiling and testing maps," on page 83.
- Activity Editor has replaced Code Editor and is described in Chapter 5, "Customizing a map," on page 103.
- Relationship Designer has changes to its interface. These changes are described in Chapter 7, "Creating relationship definitions," on page 237.
- Support is provided for a new transformation: Cross-Reference. It is described in Chapter 2, "Creating maps," on page 15.
- Map Designer and Relationship Designer now connect to System Manager (instead of InterChange Server directly), with no more login process to any server and no more saving to or opening from the server.
- A New Map wizard has been added. It is described in Chapter 2, "Creating maps," on page 15.
- Projects have been incorporated into menus and dialogs.
- A new map property--Mapping role--has been added. It is described in Chapter 3, "Working with maps," on page 55.
- Debugging a map has been improved, as has error identification in Java code.
- Better support is provided for verb and polymorphic mapping.
- More options are available for map validation.
- The GUI is improved for message definition.
- Better support is provided for upgrading to a new map format.
- Information on the getIntMilliseconds() method has been added to the DtpDate class.
- Information on the getLocale() and setLocale() methods has been added to the BusObj class.

---

## New in release 4.1.1

This product has been internationalized.



---

## New in release 4.1.0

The changes made in CrossWorlds 4.1.0 do not affect the content of this document.

---

## New in release 4.0.1

The changes made in CrossWorlds 4.0.1 do not affect the content of this document.

---

## New in release 4.0.0

- Map Designer has a new interface design. This new interface is described in the following chapters:
  - Chapter 2, “Creating maps,” on page 15
  - Chapter 3, “Working with maps,” on page 55
  - Chapter 4, “Compiling and testing maps,” on page 83
  - Chapter 5, “Customizing a map,” on page 103
- Map Designer no longer stores the map design in map design (.dlm) files. Map content is now stored in the repository in Extended Markup Language (XML) format. When you open a map that includes a .dlm file, Map Designer asks you if you want to convert it. For more information, see “Opening a map” on page 55.
- The Mapping API has been extended to support a more flexible way to obtain a connection to an external database from within a map: database connection pools. The methods in two new classes, `CwDBConnection` and `CwDBStoredProcedureParam`, replace the methods in the existing classes `DtpConnection` and `UserStoredProcedureParam`.

In support of this feature, the document has the following changes:

- New chapters for new classes:
  - Chapter 13, “`CwDBConnection` class,” on page 365
  - Chapter 14, “`CwDBStoredProcedureParam` class,” on page 377
- New information on how to obtain a connection from a database connection pool, “Executing database queries” on page 203
- Deprecation of information on the old way to obtain a connection:
  - Chapter 15, “`DtpConnection` class,” on page 381
  - Chapter 26, “`UserStoredProcedureParam` class,” on page 487
  - “Executing queries in the relationship database” on page 299
- Map initiators have been enhanced to provide more functionality.
  - A *map initiator* has been renamed to a *calling context* to better identify its purpose.
  - Existing map initiators have been renamed to better identify the context with which each is associated:

---

Old Map-Initiator Constant	New Calling-Context Constant
SUBSCRIPTION_DELIVERY	EVENT_DELIVERY
CONSUME	SERVICE_CALL_REQUEST
DELIVERBUSOBJ	SERVICE_CALL_RESPONSE
CONSUME_FAILED	SERVICE_CALL_FAILURE
ACCESS_RETURN_REQUEST	ACCESS_RESPONSE

---

- A new calling context has been provided for an access client's initiation of a flow: ACCESS\_REQUEST. Relationship methods in the Mapping API perform the same tasks for this new calling context as for the EVENT\_DELIVERY calling context.

For more information, see "Calling contexts" on page 190.

- InterChange server no longer supports Mercator maps. Therefore, the appendix that provided information on Mercator maps has been removed from this manual.
- Relationship Designer has been enhanced to allow you to cache the relationship tables associated with a static relationship. For more information, see "Optimizing a relationship" on page 253.

---

## Part 1. Maps



---

## Chapter 1. Introduction to map development

This chapter provides an overview of data mapping, introduces the tools you use to implement maps, and describes map and relationship definitions.

This chapter covers the following topics:

- “About data mapping” on page 3
- “Maps: A closer look” on page 5
- “Tools for map development” on page 7
- “Overview of map development” on page 11

---

### About data mapping

*Data mapping* is the process of transforming (or mapping) data from one application-specific format to another. Mapping is central to the process of transferring information between different applications, and for providing collaborations (business processes) that are independent of specific applications. By mapping data between application-specific business objects and generic business objects, WebSphere creates the environment that allows for the use of “best of breed” applications. The WebSphere business integration system provides a modular and extensible architecture for easy maintenance of your maps.

The WebSphere map development system provides comprehensive support for mapping between business objects, including the following capabilities:

- Transforming data values from one or more attributes in a source business object to one or more attributes in a destination business object
- Establishing and maintaining relationships between data entities that are equivalent but are represented differently and cannot be directly transformed
- Enabling access to external mapping resources, such as third-party mapping products and databases for performing queries

When data mapping is set up among differing applications, an event occurrence in one application is performed in any other application to which it is mapped. An event occurrence can be when data is created, retrieved, updated, or deleted.

Mapping uses *maps* that define the transfer (or transformation) of data between the source and destination business objects. In the map development environment, data is mapped from an application-specific business object to a generic business object or from a generic business object to an application-specific business object. Table 1 lists the types of mapping required.

*Table 1. Mapping requirements*

Direction of business object	Source business object	Destination business object	Type of map
Connector to collaboration	Application-specific	Generic	Inbound map
Collaboration to connector	Generic	Application-specific	Outbound map

**Example:** Figure 1 illustrates how mapping occurs at run time, using a fictionalized Employee Management collaboration as an example.

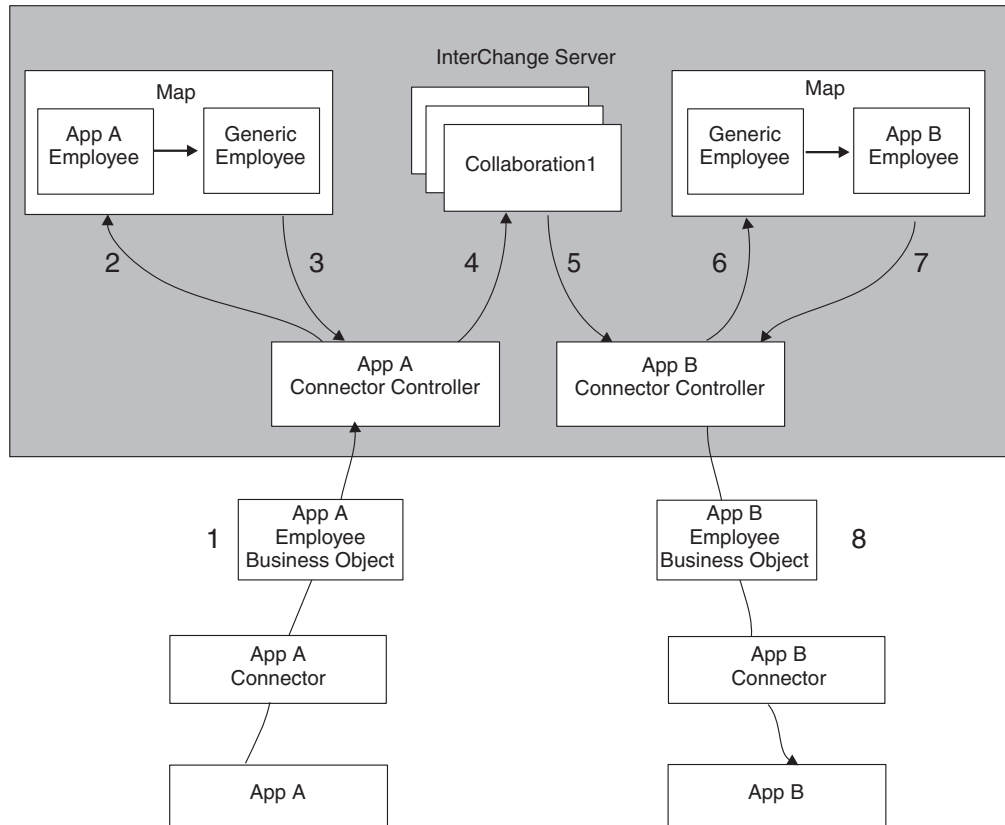


Figure 1. Data mapping at run time

The Employee Management collaboration (Collaboration1) receives an Employee business object from the source connector (App A), then sends an Employee business object to the destination connector (App B). Figure 1 illustrates the following sequence occurs (the numbers here correspond to the numbers in the figure):

1. An event occurs in App A. The App A connector produces an App A Employee business object and sends it to the App A connector controller.
2. The App A connector controller sends the App A Employee business object to the Employee Management collaboration (Collaboration1), which resides on InterChange Server, for mapping. The request includes the name of the data map that the server must use, based on the map name specified in the connector configuration.
3. The inbound map returns the generic Employee business object to the App A connector controller.
4. The App A connector controller checks the collaborations that have subscriptions to the generic Employee business object. In this case, Collaboration1 has a subscription, so the connector controller hands the business object to Collaboration1.
5. The collaboration performs some processing, then produces another generic Employee business object as output, which it sends to the App B connector controller.
6. The App B connector controller sends the generic business object to InterChange Server, requesting mapping to the App B Employee business object.

7. The outbound map returns the application-specific Employee business object to the App B connector controller.
8. The App B connector controller passes the App B Employee object to the App B connector, which can then pass the data in the business object into App B.

The figure shows two types of maps in use:

- One inbound map from the App A Employee business object to the generic Employee business object used by the collaboration
- One outbound map from the generic Employee business object to the App B Employee business object

The Employee data moves in only one direction—from Application A toward Application B. If you want to exchange the Employee data in both directions between both applications, two more maps are required:

- An inbound map from the application-specific business object of Application B to the generic business object
- An outbound map from the generic business object to the application-specific business object of Application A

---

## Maps: A closer look

As Table 2 shows, a map is a two-part entity, consisting of a map definition and a run-time object.

### Map definition

You define a map to the map development system with a *map definition*. Map definitions are stored in projects in System Manager. The Map Designer tool provides dialogs to assist in the creation of the map definitions (often referred to simply as maps). It also handles storing the completed map definition in projects in System Manager.

For more information on how to use Map Designer to create map definitions, see “Creating a map: Basic steps” on page 30..

The map definition provides the following information about the map:

- The map name
- The source and destination objects of the map
- The map transformations

### Map definition name

A map definition is simply a template or description of the map. It provides information on how to transform attributes of one business object to another. Therefore, the name of the map definition should identify the direction of the map and the business objects it transforms.

### Source and destination business objects

Maps consist of one or more source business objects and one or more destination business objects. The *source business objects* are the ones to be transformed; the *destination business objects* are the ones that are generated with data from the source business objects.

### Map transformations

The rest of the map consists of a series of transformation steps. A *transformation step* is a segment of Java code that returns the value of a destination attribute. A

map contains one transformation step for each destination attribute that is transformed. Transformations are implemented as Java code and are therefore stored in a Java source (.java) file.

Table 2 shows some of the transformations you can perform on a destination business object. Standard transformations include Set Value, Move, Join, Split, Submap, and Cross-Reference. You can create custom transformations with graphical function blocks, as well as with Java code for "Relationships," "Content-based logic," "Date Conversion," and "String transformations."

*Table 2. Transformations of a map*

<b>Transformation</b>	<b>Description</b>	<b>For more information</b>
Standard transformations	Transformations for which Map Designer can autogenerate code	
Set Value	Specifying a value for a destination attribute	"Specifying a value for an attribute" on page 37
Move (Copy)	Copying a source attribute to a destination attribute	"Copying a source attribute to a destination attribute" on page 38
Join	Joining two or more source attributes into a single destination attribute	"Joining attributes" on page 39
Split	Splitting a source attribute into two or more destination attributes	"Splitting attributes" on page 41
Submap	Calling a map for a child business object	"Transforming with a submap" on page 43
Cross-Reference	Maintaining identity relationships for the business objects	"Cross-referencing identity relationships" on page 47
Custom transformations	Creating a transformation other than one of the standard transformations listed above	"Creating a Custom transformation" on page 47
Relationship	Associating business objects that cannot be directly mapped because each application maintains the data in its own format	Chapter 8, "Implementing relationships," on page 257
Content-based logic	Transforming a destination attribute based on the content of the source attribute	"Content-based logic" on page 174
Date conversion	Converting a date from its format in the source attribute to its format in the destination attribute	"Date formatting" on page 179
String	Performing basic transforms on a string, such as case conversion and obtaining substrings	"Using Expression Builder for string transformations" on page 182

When a clear correspondence exists between the source attribute and destination attribute, the transformation step simply copies the source value to the destination attribute. Other transformations can involve calculations, string manipulations, data type conversions, and any other logic that you can code using Java.



Figure 2 illustrates some typical kinds of attribute transformations:

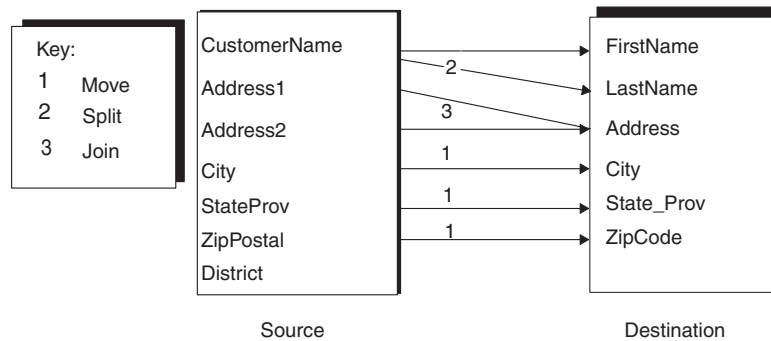


Figure 2. Typical attribute transformations

As Figure 2 shows, attributes from the source business object are typically:

- Copied to a destination attribute (City, StateProv, ZipPostal).
- Split into multiple destination attributes (CustomerName).
- Joined into one destination attribute (Address1, Address2).
- Ignored when the destination object has no equivalent attribute (District).

For simple transformations such as copying a value into an attribute, splitting a value into two or more attributes, or joining two or more values into one attribute, you can specify the step graphically and Map Designer generates the Java code. For more complex transformations, you can customize the transformation with a graphical editor or write your own Java code.

## Map instance

The map definition is a template for the run-time instantiation of the map, the *map instance*. During map execution, the Map Development system creates instances of the map based on the map definition and the transformation code.

Each map instance provides the following information:

- Basic functionality such as logging, tracing, connections, and exception handling through methods of the BaseDLM class
- The map execution context

For more information, see “Understanding map execution contexts” on page 189.

A map instance is represented in the Mapping API by an instance of the BaseDLM class.

---

## Tools for map development

Table 3 shows the two graphical design tools of mapping.

Table 3. Principal components of data mapping system

Design tool	Mapping component	Description
Map Designer	Map	Uses Java code to specify how to transform attributes from one or more source business objects to one or more destination business objects. You typically create one map for each source business object you want to transform, though you can also break up a map into several submaps.
Relationship Designer	Relationship	Establishes an association between two or more data entities in the Map Development system. Relationship definitions most often associate two or more business objects. You use relationship definitions to transform data that is equivalent across business objects but is represented differently. For example, a state code for the state of Michigan might be represented as MI in one application and MICH in another. This data is equivalent but is represented differently in each application. Most maps use one, or a few, relationship definitions.

These graphical tools run on Windows 2000 and Windows XP. Therefore, these platforms are for map development.

Table 4 lists the additional tools that are supported for map development.

Table 4. Tools for map development

Tool	Description
Mapping API	Set of Java classes with which you can customize the generated mapping code.
System Manager	Provides graphical windows to configure a map instance as well as configure a relationship object.

## Map Designer

Map Designer creates and compiles maps. You can launch Map Designer from System Manager by selecting Map Designer from the Tools menu. For other ways to launch Map Designer, see “Starting Map Designer” on page 15.. Map Designer provides a tab window to view map information. This window displays one of four tabs: Table tab, Diagram tab, Messages tab, or Test tab.

Figure 3 shows a map displayed in the Diagram tab of Map Designer.

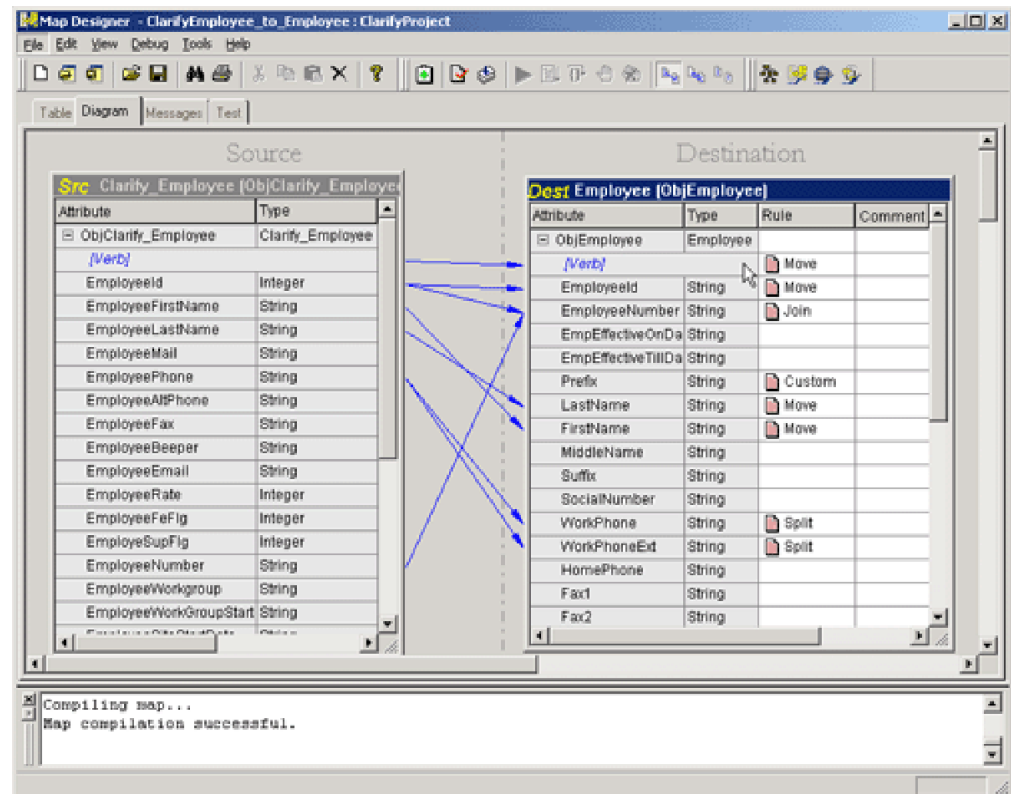


Figure 3. Map Designer

For information on how to use Map Designer to create a map, see Chapter 2, “Creating maps,” on page 15.

## Relationship Designer

Relationship Designer creates relationship definitions that store the run-time relationship instance data. You can launch Relationship Designer from System Manager by selecting Relationship Designer from the Tools menu. Figure 4 shows

several relationships displayed in Relationship Designer.

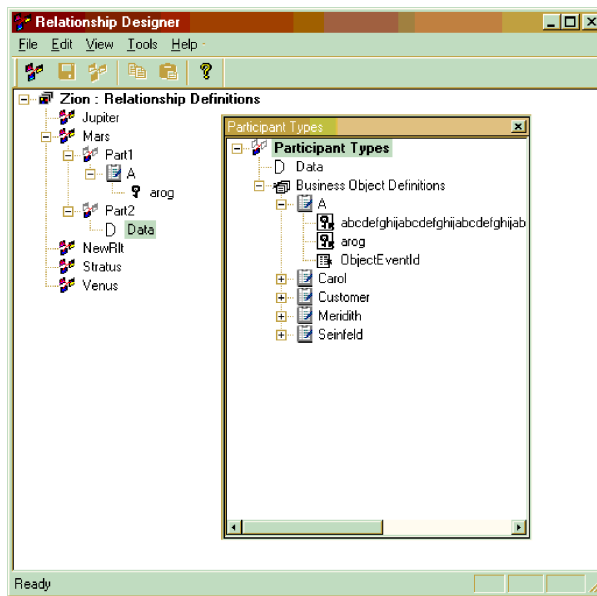


Figure 4. Relationship Designer

For more information on how to use Relationship Designer, see Chapter 7, “Creating relationship definitions,” on page 237.

## Mapping API

Many transformation steps can be programmed using standard Java methods. To make writing transformation steps easier, the map development system provides a mapping API (described in detail in Part 3, “Mapping API Reference,” on page 313), with methods to handle the most common data transformation situations. The mapping API includes the following classes:

- DTP (Data Transformation Package) classes provide methods for string manipulation, data type conversion, date manipulation, submap calling, and SQL query execution. The classes are:
  - DtpConnection (deprecated)
  - DtpDataConversion
  - DtpDate
  - DtpMapService
  - DtpSplitString
  - DtpUtils
- Business object classes are used for both collaboration development and mapping. The classes are:
  - BusObj
  - BusObjArray
- Relationship management classes provide methods for creating and managing relationship instances. The classes are:
  - Participant
  - Relationship
  - IdentityRelationship

- Database connection classes provide methods for SQL query execution. These classes are:
  - CwDBConnection
  - CwDBStoredProcedureParam
  - DtpConnection (deprecated)
  - UserStoredProcedureParam (deprecated)
- Utility classes assist with error handling and debugging, and setting important run-time values for maps. The classes are:
  - BaseDLM
  - MapExeContext

## System Manager

System Manager is a graphical tool that provides an interface to InterChange Server and the repository. System Manager provides the means to manage maps and configure a map definition. You can:

- Set some general properties of a map definition, including its trace level and data validation level.
- Display the source and destination business objects of a map.
- Compile a map definition.

**Note:** System Manager provides ways to start up Map Designer. For more information, see “Starting Map Designer” on page 15.

System Manager also provides the means to manage relationships. You can:

- Set some general properties of a relationship, including the location of its relationship tables.
- Display the participants of the relationship.

**Note:** System Manager also provides ways to start up Relationship Designer. For more information, see “Starting Relationship Designer” on page 237.

---

## Overview of map development

This section provides an overview of map development, which includes the following high-level tasks:

1. Installing and setting up the map development software and installing the Java Development Kit.
2. Designing and implementing the map.

## Setting up the development environment

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

- The map development software is installed on a machine that you can access. For information on how to install and start up the map development software system, see your system installation guide.
- The IBM Java Development Kit (JDK) is installed from the product CD. Be sure to update the PATH environment variable to include the installed Java directory. Restart InterChange Server after you have updated the path.
- System Manager is running.

For information on starting up System Manager, see your system installation guide.

- Map Designer is open and connected to System Manager.

For information on how to start Map Designer, see “Overview of Map Designer” on page 15.

## Designing and implementing the map

To design and implement maps you need to do the following:

1. Learn the data formats used by all business objects involved in the map.
2. Create the map within Map Designer.
3. Customize any required transformation rule.
4. Define any relationships within Relationship Designer that the map needs.
5. Customize the mapping transformation to perform relationship management.
6. Implement error and message handling, if appropriate.
7. Generate the .java file and compiled code. The compiled code is an executable Java class. For more information, see “Map development files” on page 13.
8. Test and debug the map, recoding as necessary.

Figure 4 provides a visual overview of map development and provides a quick reference to chapters where you can find information on specific topics.

**Tip:** If a team of people is available for map development, the major tasks of developing a map can be done in parallel by different members of the development team.

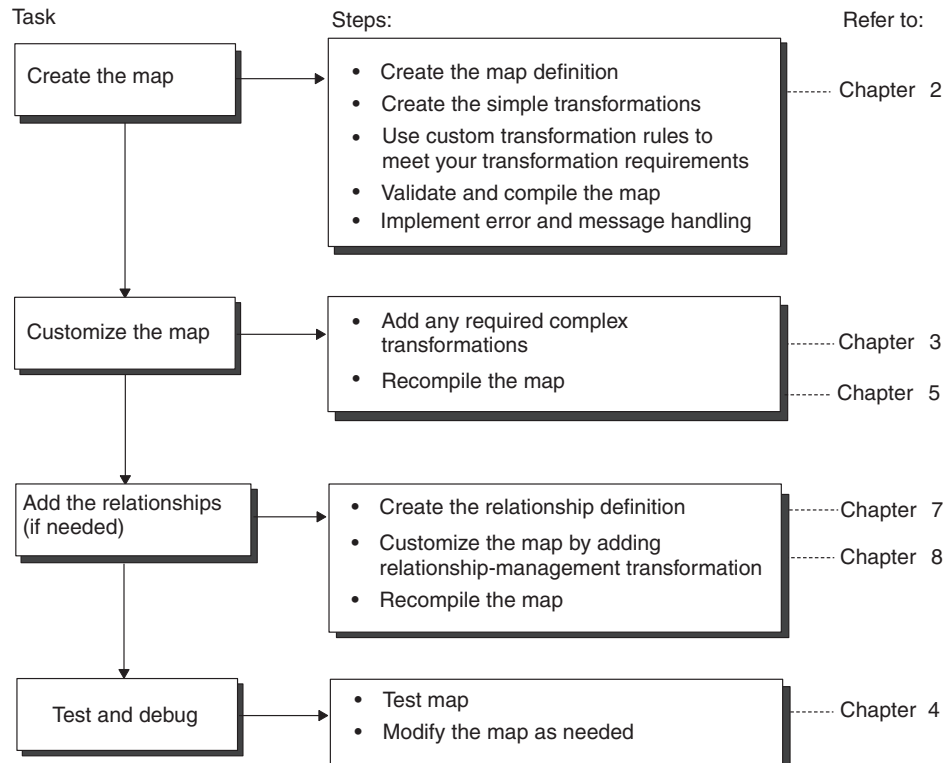


Figure 5. Overview of the map development task

## Map development files

The following information forms the basis of the map:

- When you compile a map, Map Designer generates two types of files (.java, .class) or an optional message file (.txt) if map-specified messages are defined in the map. These files are saved in the project in System Manager.
- Map Designer generates a map definition when you save a map to the project in System Manager. This map definition contains general information about the map (such as map properties) as well as information about how the destination attributes are mapped.

**Attention:** Do not modify the *mapname.java* file. If you do, your changes are not reflected in the map design, which is stored in the project in System Manager. Therefore, these changes are not editable in Map Designer. Map Designer reads only the map definition.

Relationship Designer also stores relationship definitions in XML format in System Manager. At deployment, System Manager creates table schemas in the relationship database to contain the relationship run-time instance data. For each relationship, you can specify the location of all its relationship tables. The default location for these tables is the IBM WebSphere Business Integration Server repository.

Table 5 lists the file types that Map Designer can generate (.java, .class, .cwm, .bo, .txt) and their locations relative to the System Manager workplace.

Table 5. Map file types

File type	Description	Location relative to System Manager workspace
.java	Generated Java code, created by Map Designer when you compile a map.	Stored in ProjectName\Maps\Src.
.class	Compiled Java code, created by Map Designer when you compile a map.	Stored in ProjectName\Maps\Classes.
.cwm	Map definition file, generated by Map Designer when you save a map definition.	Saved to ProjectName\Maps when "Saved" to System Manager.
.bo	Plain text file, used to save and load test run data and to save test run results.	You can save these files to any location.
.txt	Message file, created by Map Designer from information in the Messages tab when it compiles the map.	Stored in ProjectName\Maps\Messages.





---

## Chapter 2. Creating maps

This chapter provides an overview of Map Designer and describes how to use Map Designer to create maps.

**Note:** This chapter frequently uses the terms *map* and *map definition* interchangeably. When the term *map* is used, it refers to the map definition (what is accessed through Map Designer).

This chapter covers the following topics:

- “Overview of Map Designer” on page 15
- “Creating a map: Basic steps” on page 30
- “Mapping standards” on page 52

For background information on how the WebSphere business integration system uses maps, see Chapter 1, “Introduction to map development,” on page 3.

---

### Overview of Map Designer

Map Designer is a graphical development tool for creating and modifying maps. A *map* is made up of a series of transformation steps that define how to calculate the value for each attribute in the destination business object. Creating a map is the process of specifying the transformation steps for each destination attribute that you want to transform.

Using Map Designer, you can specify simple transformation steps, such as copying a source attribute to a destination attribute of the same data type, interactively using drag-and-drop. Map Designer automatically generates the Java code necessary to perform the transformation.

To assist with other common transformations, such as splitting a source attribute into multiple destination attributes or joining multiple source attributes into a single destination attribute, Map Designer prompts you for information, such as the delimiter on which to split or join, then generates the necessary Java code. To specify more complex transformations, you can define activities graphically using Activity Editor in a custom transformation rule, modify the Java code directly in the Activity Editor window, or write your own transformation steps from scratch.

This section covers the following topics to introduce you to Map Designer:

- “Starting Map Designer” on page 15
- “Working in projects” on page 16
- “Layout of Map Designer” on page 16
- “Assigning preferences” on page 21
- “Customizing the main window” on page 24
- “Using Map Designer functionality” on page 25

### Starting Map Designer

To launch Map Designer, do one of the following:

- From System Manager, perform one of these actions:

- From the Tools menu, select Map Designer.
- Click a map folder in a project to enable the Map Designer icon in the System Manager toolbar. Then click the Map Designer icon.
- Right-click the map folder in a project and select Create New Map from the Context menu.
- Right-double-click a map to start Map Designer with the selected map opened.
- From a development tool, such as Business Object Designer or Relationship Designer, perform one of these actions:
  - From the Tools menu, select Map Designer.
  - In the Programs toolbar, click the Map Designer button.
- Use a system shortcut:  
 Start > Programs > IBM WebSphere InterChange Server > Toolset > Development > Map Designer

**Important:** For Map Designer to be able to access maps stored in System Manager, Map Designer must be connected to an instance of System Manager. The preceding steps assume that you have already started System Manager. If System Manager is already running, Map Designer will automatically connect to it.

Map Designer displays in its own application window. You can launch more than one instance of Map Designer at a time to edit more than one map.

## Working in projects

Map Designer views, edits, and modifies maps stored in System Manager on a *project* basis. A *project* is simply a logical grouping of entities for management and deployment purposes. System Manager allows you to create multiple projects.

When Map Designer establishes a connection to System Manager, it obtains a list of business objects that are defined in the current project. If you add or delete a business object using Business Object Designer, System Manager notifies Map Designer, which dynamically updates the list of business object definitions.

Before you can work on a map, you need to select which project the map is in by entering the name of the project in the Open a Map from a Project dialog. Before you switch to another project, you need to save the maps you modified in the current project. For more information on opening a map from a project and saving a map in a project, see “Steps for opening a map from a project in System Manager” on page 56 and “Saving a map to a project” on page 49, respectively.

## Layout of Map Designer

When you first open Map Designer without specifying a map, the Map Designer tab window is empty and the output window does not display. When you open an existing map, the Map Designer window displays the Map tabs in the tab window.

Table 6 describes each of the components in the Map Designer main window.

Table 6. Components of the Map Designer window

Window area	Description	For more information
Menus	Provide options to access Map Designer functionality.	“Map Designer pull-down menus” on page 25

Table 6. Components of the Map Designer window (continued)

Window area	Description	For more information
Toolbar	Actually contains three separate toolbars, each of which provides a set of buttons to access Map Designer functionality.	"Map Designer toolbars" on page 28
Map Designer tab window	Displays map information for an open map in one of four Map tabs.	"Table tab" on page 17 "Diagram tab" on page 19 "Messages tab" on page 20 "Test tab" on page 20
Output Window	Displays results from the compilation of a map and other status messages. If the output window is not currently displaying when Map Designer generates a status message, it opens this window automatically. You can clear the contents of the output window with the Clear Output option of the View menu.  <b>Tip:</b> You can control whether the output window pane displays as part of the main window of Map Designer with the Output Window option of the View menu.	N/A
Status Bar	Displays Map Designer status messages.  <b>Tip:</b> You can control whether the status bar displays as part of the Map Designer window with the Status Bar option of the View menu.	N/A

The following sections describe the general layout of each of the tabs that display in Map Designer's tab window.

### Table tab

The Table tab of Map Designer displays mapping information in a tabular format that lists all mapping attributes and transformations.

The Table tab consists of the following areas:

- Attribute Transformation Table
- Business Objects Pane

**Attribute Transformation Table:** The attribute transformation table presents in a tabular format all transformations associated with the map. Table 7 shows the columns that make up this table.

Table 7. Columns of the Attribute Transformation Table

Column name	Description
Exec. Order	The execution order for the destination attribute.  When you add a transformation to the end of this table, Map Designer automatically assigns its execution order as the last in the table. You can change the execution order of an attribute by typing the desired order number in the Exec. Order field.  <b>Note:</b> You can specify how Map Designer handles the execution order of destination attributes with the option <i>Defining Map: automatically adjust execution order</i> . By default, this option is disabled. When the option is enabled, Map Designer automatically adjusts the execution order of other attributes. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see "Specifying General Preferences" on page 22.

Table 7. Columns of the Attribute Transformation Table (continued)

Column name	Description
Source Attribute	<p>The name of the source attribute for the transformation.</p> <p>This field provides a combo box that contains a list of all source and destination business objects with their attributes listed under them. Click the appropriate source attribute from this list. You can select multiple source attributes by clicking the Multiple Attributes entry in the combo box list. Map Designer displays the Multiple Attributes dialog from which you can select the attributes.</p> <p><b>Note:</b> You can specify how Map Designer displays the source attribute name with the option <i>Defining Map: show full attribute path</i>. By default, this option is disabled and Map Designer displays all source attribute names as <i>...AttrName</i>. When the option is enabled, Map Designer displays the full attribute path: <i>ObjSrcBusObj.AttrName</i>. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.</p>
Source Type	<p>The data type of the source attribute.</p> <p>This field is read-only.</p>
Destination Attribute	<p>The name of the destination attribute for the transformation.</p> <p>This field provides a combo box that contains a list of all source and destination business objects with their attributes listed under them. Click the appropriate destination attribute from this list.</p> <p><b>Note:</b> You can specify how Map Designer displays the destination attribute name with the option <i>Defining Map: show full attribute path</i>. By default, this option is disabled and Map Designer displays all destination attribute names as <i>...AttrName</i>. When the option is enabled, Map Designer displays the full attribute path: <i>ObjDestBusObj.AttrName</i>. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.</p>
Dest. Type	<p>The data type of the destination attribute.</p>
Transformation Rule	<p>This field is read-only.</p> <p>The transformation rule and code for this attribute’s transformation step.</p> <p>This field provides a combo box that contains a list of standard transformations:</p> <ul style="list-style-type: none"> <li>• None (no transformation)</li> <li>• Join</li> <li>• Move</li> <li>• Split</li> <li>• Set Value</li> <li>• Submap</li> <li>• Cross-Reference</li> <li>• Custom</li> </ul> <p>Click the appropriate transformation from this list to enter it in the field. For more information, see “Specifying standard attribute transformations” on page 36.</p>
Comment	<p>An informational description of the attribute’s transformation.</p> <p>See “Setting comments in the comment field of the attribute” on page 52.</p>

**Steps for defining a map from the Table tab:** To define a map from the Table tab, follow these general steps:

1. Click in an empty cell in the Source Attribute column. From the available combo box, click the source attribute to transform.

2. Click in the corresponding cell in the Destination Attribute column. Click the destination attribute from the available combo box.
3. Click in the corresponding cell in the Transformation Rule column. This column provides a combo box:
  - For a standard transformation (Join, Move, Split, Set Value, Submap, or Cross-Reference), select the associated option from the list. Map Designer generates code for these standard transformations. You can customize this code as needed. For more information, see “Specifying standard attribute transformations” on page 36.
  - For a transformation that is *not* in this combo box, select Custom from the list and add the custom Java code in Activity Editor. For more information, see “Creating a Custom transformation” on page 47.
4. Click in the corresponding cell in the Comment column. For more information, see “Setting comments in the comment field of the attribute” on page 52.

**Business Objects Pane:** The business objects pane presents in a list all source and destination business objects associated with the map. Its left area displays the source business objects; its right area displays the destination business objects. If the map contains a temporary business object, the business objects pane contains three areas: Source Business Object, Temporary Business Object, and Destination Business Object.

**Tip:** You can control whether the business objects pane displays as part of the Table tab with the Business Objects Pane option of the View menu.

## Diagram tab

The Diagram tab of Map Designer provides a drag-and-drop interface for defining and reviewing the transformations. You view and design maps in the map workspace, which displays on the right side of the window.

The Diagram tab consists of the following areas:

- Business object browser, which displays in the project pane, on the leftmost part of the window. This browser uses a hierarchical format to list the business objects in the project in System Manager when Map Designer is connected to System Manager. To refresh the list of business objects in the business object browser, right-click in the business object browser and select Refresh All from the Context menu. Map Designer queries System Manager and updates the business object browser with the current business objects.

**Note:** If you add or delete a business object from the project in System Manager, System Manager dynamically updates the list of business object definitions.

**Tip:** You can control whether the business object browser displays as part of the Diagram view with the Project Pane option of the View pull-down menu.

- Map workspace, which always displays the information about the current map. When you open a map, the map workspace displays a business object window for each source and destination business object used in the map. Each business object window lists some or all attributes defined in the business object, depending on what viewing mode is currently selected. In the case of a destination business object or temporary business object, the business object window also lists the transformation rule and comments associated with the attribute. In the map workspace, you can add, delete, or modify transformations in the map. Lines connecting attributes represent the transformations between the attributes.

**Tip:** You can control which attributes display in the source and destination business objects in the Diagram tab with the options of the View > Diagram submenu. This submenu allows you to select whether to display all attributes, only linked (mapped) attributes, or only unlinked (unmapped) attributes.

## Messages tab

The Messages tab displays the map's messages. A message consists of a message ID and its associated message text.

The Messages tab is divided into two panes. The top pane is the message grid, which consists of three columns: Message ID column, Message column, and Explanation column (for comments for the entire message file). The bottom or Description pane is for entering plain text. When you enter text into the Description pane, the text is added to the top of the generated message file as comments. Map Designer saves any change made to the map's messages in the project of System Manager.

For more information on messages and how to use them, see Appendix A, "Message files," on page 497. For information about the format of messages, see "Format for map messages" on page 500.

When you compile a new map, Map Designer generates an external message file, based on the information entered in the Messages tab. This message file is saved in the message directory.

**Attention:** You must make all changes to a map's messages through the Messages tab of Map Designer. Do *not* use an external text editor to make changes to the generated message file. Any changes made from the external editor will *not* be visible to Map Designer because they will *not* be stored in the map definition of the project. Furthermore, such changes will be overwritten the next time you compile the map.

## Test tab

The Test tab provides an interface for testing maps and viewing the results. In this tab, you can run tests to verify that transformations are working properly.

The Test tab consists of the following areas:

- Test path diagram

The test path diagram at the top of the window shows the map test as a series of icons:

- The Source Testing Data arrow indicates the direction of the map transformation and is labeled with the business object type for the source business object that is participating in the map test.
- The Map icon represents the currently open map, which is used in the test.
- The Destination Testing Data arrow indicates the direction of the map transformation and is labeled with the business object type for the destination business object that results from the map test.

- Source Testing Data pane

The source testing data area in the lower left window uses a hierarchical format to list the attributes of the source business object that participates in the map. Click the plus symbol (+) next to a source business object to expand it. In this area, you enter test data for the source business object.

- Destination Testing Data pane

The destination testing data area in the lower right window uses a hierarchical format to list the attributes of the destination business object that results from the map. Click the plus symbol (+) next to a business object to expand it. In this area, you view test results data for the destination business object.

**Note:** Map Designer displays results from the test run of the map in the output window.

For more information on how to use the Test tab, see “Testing maps” on page 87..

## Assigning preferences

The Preferences dialog allows you to customize the behavior of the Map Designer tool. To display the Preferences dialog:

- From the View menu, select Preferences.
- Use the keyboard shortcut of `Ctrl+U`.

Figure 6 shows the Preferences dialog.

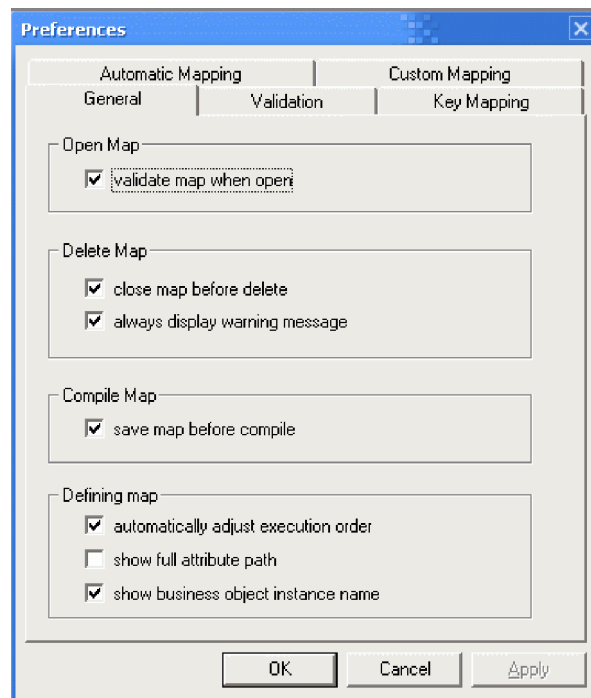


Figure 6. Preferences dialog

Map Designer saves preference settings in the Windows registry. Therefore, they remain in effect for the current Map Designer session and future sessions. The Preferences dialog provides the following tabs:

- General
- Validation
- Key Mapping
- Automatic Mapping
- Custom Mapping



## Specifying General Preferences

The General tab of the Preferences dialog displays the general preferences you can specify for how Map Designer manages maps.

Table 8. General Map Designer Preferences

General Preference	Description	For more information
Open Map		
validate map when open	When this option is enabled, Map Designer validates the map when it opens it.  <b>Recommendation:</b> If a map uses business objects with many attributes, that is, more than a thousand attributes, enabling this option may result in the map taking a long time to open. If that is the case, and it is not desirable, you should disable this option.	“Opening a map” on page 55
Delete Map		
close map before delete	When this option is enabled, Map Designer always closes the currently open map before displaying the Delete Map dialog.	“Steps for deleting maps” on page 76
always display warning message	When this option is enabled, Map Designer always displays a confirmation before deleting a map.	“Steps for deleting maps” on page 76
Compile Map		
save map before compile	When this option is enabled, Map Designer always saves the current map to the project in System Manager before compiling it.	“Compiling a map” on page 84
Defining Map		
automatically adjust execution order	When this option is enabled, Map Designer automatically renumbers the execution order of destination attributes in the Table tab when execution order of an existing attribute changes.	“Using execution order” on page 77
show full attribute path	When this option is enabled, Map Designer shows the full attribute path for the names of source and destination attributes in the Table tab.	“Table tab” on page 17
show business object instance name	When this option is enabled, Map Designer displays the names of the source and destination business object <i>and</i> their variable names. When this option is disabled, Map Designer omits the names of the business object variables in both the Table and Diagram tabs.	“Steps for modifying business object variables” on page 170

## Specifying Validation

The Validation tab of the Preferences dialog provides options you can select for Map Designer to perform validations on the map when you save the map. The options are as follows:

- Show warning if verb not mapped
- Show warning if key attribute not mapped
- Show warning if required attribute not mapped
- Show warning if child business object not mapped

Map Designer will do the selected validation as deep as there are other transformation rules in that level.



**Example:** If path a.b.c is mapped, then Map Designer will perform these validations on business objects level a, a.b, and a.b.c.

For more information, see “Validating a map” on page 84.

## Specifying Key Mapping

The Key Mapping tab of the Preferences dialog displays the key mappings for several standard transformations in the Diagram tab.

Table 9. Key Mapping Map Designer Preferences

Key map	Description	For more information
Move/Join/Submap	<p>Key map to use when creating a Move, Join, or Submap transformation. Map Designer distinguishes between the transformations by the type and number of source attributes:</p> <ul style="list-style-type: none"> <li>• Move—one source attribute that is <i>not</i> a child business object</li> <li>• Join—more than one source attribute that is <i>not</i> a child business object</li> <li>• Submap—one or more source attributes that are a child business object</li> </ul>	<p>“Copying a source attribute to a destination attribute” on page 38</p> <p>“Joining attributes” on page 39</p> <p>“Transforming with a submap” on page 43</p>
Split/Cross-Reference	<p>Key map to use when creating a Split transformation or for maintaining identity relationships</p>	<p>“Splitting attributes” on page 41,</p> <p>“Cross-referencing identity relationships” on page 47</p>
Custom	<p>Key map to use when creating a Custom transformation.</p>	<p>“Creating a Custom transformation” on page 47</p>

The Key Mapping tab provides the following functionality:

- To change a key mapping, click in the appropriate transformation field and select the desired key map for this transformation from the combo box. Click OK.
- To return key mappings to their default values, click Use Default and then click OK.

## Specifying Automatic Mapping

The Automatic Mapping tab of the Preferences dialog provides options you can select for Map Designer to use when searching for matching attribute names in business objects for map automation. The options are as follows:

- Ignore Case—to perform case-insensitive name matches on the search string
- Ignore Incompatible Data types—to perform name matches with incompatible data types on the search string

**Note:** Selecting this option may result in data loss.

For more information, see “Using map automation” on page 65.

## Specifying Custom Mapping

The Custom Mapping tab of the Preferences dialog allows you to configure the standard function blocks to be used directly in Map Designer.

For more information, see “Tip: Using function blocks directly in Map Designer” on page 112

## Customizing the main window

Map Designer allows you to customize its main window by:

- “Selecting how windows display”
- “Floating a dockable window” on page 25

### Selecting how windows display

When you first open Map Designer without specifying a map, the main window is empty with the toolbars and status bar visible. When you open a map, Map Designer displays the Diagram tab in the tab window and opens the output window. By default, Map Designer displays each of the map tabs as follows:

- Table tab—the business objects pane displays under the attribute transformation table.
- Diagram tab—the map workspace area displays and is empty.
- Messages and Test tabs—as described in “Messages tab” on page 20 and “Test tab” on page 20, respectively.

You can customize the appearance of the main window and the Map tabs with options from the View menu. Table 10 describes the options of the View pull-down menu and how they affect the appearance of the Map Designer window.

*Table 10. View menu options for Map Designer window customization*

View menu option	Element displayed
Toolbars	A submenu with options for each of the Map Designer toolbars: <ul style="list-style-type: none"><li>• Standard toolbar</li><li>• Designer toolbar</li><li>• Programs toolbar</li></ul>
Status Bar	A single-line pane in which Map Designer displays status information.
Business Objects Pane	A pane that displays the source and destination business objects in the Table tab of Map Designer.
Project Pane	A pane that displays the business object browser in the Diagram tab of Map Designer.
Diagram	A submenu with options for which attributes to display in the source and destination business objects in the business object windows of the Diagram tab: <ul style="list-style-type: none"><li>• All Attributes</li><li>• Linked Attributes</li><li>• Unlinked Attributes</li></ul> <p>The Designer toolbar also provides icons for displaying these attributes.</p>
Output Window	A small window across the bottom of the Map Designer window. The Clear Output option of the View menu clears all text in the output window.

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

## Floating a dockable window

Map Designer supports the following features as dockable windows:

- Toolbars in the main window:
  - Standard toolbar
  - Designer toolbar
  - Programs toolbar
- Output Window
- Find Control pane. For more information, see “Finding information in a map” on page 72.

**Tip:** 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.

## Using Map Designer functionality

You can access Map Designer’s functionality using any of the following:

- Pull-down menus
- Context menu
- Toolbar buttons
- Keyboard shortcuts

### Map Designer pull-down menus

Map Designer provides the following pull-down menus:

- File menu
- Edit menu
- View menu
- Debug menu
- Tools menu
- Help menu

The following sections describe the options of each of these menus.

**Functions of the File menu:** The File pull-down menu of Map Designer provides the options shown in Table 11.

Table 11. Options of the File menu in Map Designer

File menu option	Description	For more information
New	Creates a new map file, clearing any existing map from the map workspace	“Creating a map: Basic steps” on page 30
Open	Opens an existing map From Project or From File	“Opening a map” on page 55
Close	Closes the current map	“Closing a map” on page 57
Save	Saves the current map to the same name To Project or To File	“Saving maps” on page 49

Table 11. Options of the File menu in Map Designer (continued)

File menu option	Description	For more information
Save As	Saves the current map to a name different from the map To Project or To File	"Saving maps" on page 49
Delete	Deletes a specified map	"Deleting objects" on page 74
Validate Map	Validates the current map	"Validating a map" on page 84
Compile	Compiles the current map	"Compiling a map" on page 84
Compile with Submap(s)	Compiles the current map and its submaps	"Compiling a map" on page 84
Compile All	Compiles all or a subset of maps defined	"Compiling a set of maps" on page 86
Create Map Document	Creates HTML files that describe the map between business objects	"Steps for creating a map document" on page 63
View Map Document	Displays the HTML map-document file in your HTML browser	"Viewing a map document" on page 64
Print Setup, Print Preview, Print	Provides options for previewing, printing, and configuring a print job	"Printing a map" on page 74
Exit	Exits Map Designer	N/A

**Functions of the Edit menu:** The Edit pull-down menu of Map Designer provides the following options:

- Standard Windows edit options—Cut, Copy, and Paste
- Delete Current Selection—Deletes the currently selected object
- Select All—In the Diagram tab, selects all transformations between the source and destination business objects
- Insert Row—Inserts a row before the current row in the attribute transformation table of the Table tab
- Add Business Object—Displays the Add Business Object dialog to add business objects (source, destination, and temporary) to the map
- Delete Business Object—Displays the Delete Business Object dialog to delete a business object
- Find—Searches an attribute name or transformation code for text or transformation code for unmapped attributes
- Replace—Searches and replaces in custom Java code or comments
- Map Properties—Displays the Map Properties window

**Functions of the View menu:** The View pull-down menu of Map Designer provides the following display options:

- Business Objects Pane—When enabled, displays the source and destination business objects at the bottom pane of the Table tab in the Map Designer window
- Diagram—Provides options for displaying attributes in the business object windows of the Diagram tab
- Project Pane—Always enabled, displays the business object browser as the left pane of the Diagram tab in the Map Designer window
- Clear Output—Clears the contents of the output window
- Output Window—When enabled, displays status messages, including messages about opening, validating, saving, compiling, and test running the map
- Toolbars—Provides options for displaying the Map Designer toolbars: Standard, Designer, and Programs

- Status Bar—When enabled, displays a single-line status message at the bottom of the main window
- Preferences—Displays the Preferences dialog, from which you can set Map Designer preferences

For information on View menu options that control display, see “Selecting how windows display” on page 24.

**Functions of the Debug menu:** The Debug pull-down menu provides access to the debugging facilities of Map Designer. It provides the following options:

- Run Test—Connects to a server and starts the test run of a map that is opened from a project
- Continue—Continues execution after it stops at a breakpoint
- Step Over—Continues execution after it stops at a breakpoint, but stops execution before executing the next attribute
- Stop Test Run—Stops the test run of a map
- Advanced—Provides options for connecting to a server for testing a map that resides in the server (Attach) and disconnecting from a server and closing a map (Detach)
- Toggle Breakpoint—Sets a breakpoint in a map, which pauses execution just before the selected attribute’s transformation
- Breakpoints—Displays all breakpoints for the map
- Clear All Breakpoints—Clears all breakpoints in the map

For more information about the use of Map Designer testing and debugging facilities, see “Testing maps” on page 87.

**Functions of the Tools menu:** The Tools pull-down menu of Map Designer provides options to start each of the tools, including the Map Automation tools:

- Automatic Mapping
- Reverse Map
- Process Designer
- Map Designer
- Business Object Designer
- Relationship Designer

**Functions of the Help Menu:** The Help menu provides the standard Windows Help options:

- Help Topics
- Documentation
- About Map Designer

### Context menu

The Context menu is a shortcut menu that is available, by right-clicking, from numerous places, such as the transformation rule column, row header in the Table view, child business object in the source testing pane, or edit box in a dialog. A menu opens that contains useful commands, which change depending on where you click.

**Example:** Clicking in the transformation rule column opens a Context menu that provides the following options:

- **Open**—Opens the corresponding dialog box for the transformation rule, such as Join, Split, and Submap. For custom transformations, opens Activity Editor.
- **Open in New Window**—For custom transformations, opens a new instance of Activity Editor to show the detail of the transformation rule.
- **View Source**—Shows the transformation’s corresponding Java code in Activity Editor. Depending on the nature of the transformation, the code may be read-only.

**Note:** The default action when you double-click the transformation cell is Open. If Open is not available for that transformation, then a message saying that the action is not available is displayed in the status bar.

## Map Designer toolbars

Map Designer provides three toolbars for common tasks you need to perform:

- Standard toolbar
- Designer toolbar
- Programs toolbar

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.

**Tip:** To identify the purpose of each toolbar button, roll over each button with your mouse cursor.

**Standard toolbar:** Figure 7 shows the Standard toolbar.



Figure 7. Standard toolbar

The following list provides the function of each Standard toolbar button, left to right:

1. New map
2. Open
3. Save to project
4. Open from file
5. Save to file
6. Find in map
7. Print map
8. Cut
9. Copy
10. Paste
11. Delete
12. Help

**Designer toolbar:** Figure 8 shows the Designer toolbar.



Figure 8. Designer toolbar

The following list provides the function of each Designer toolbar button, left to right:

1. Add Business Object
2. Validate
3. Compile
4. Run Test
5. Continue
6. Step over
7. Toggle Breakpoints
8. Clear All Breakpoints
9. All Attributes
10. Linked Attributes
11. Unlinked Attributes

**Programs toolbar:** Figure 9 shows the Programs toolbar.



Figure 9. Programs toolbar

The following list provides the function of each Programs toolbar button, left to right:

1. Process Designer
2. Map Designer
3. Business Object Designer
4. Relationship Designer

### Keyboard shortcuts

Map Designer provides the keyboard shortcuts shown in Table 12 for many of the menu options.

Table 12. Keyboard shortcuts for Map Designer

Keyboard shortcut	Description	For more information
Ctrl+E	Save the current map definition to a map definition file	“Saving a map to a file” on page 50
Ctrl+F	Display Find control panel to locate text or unlinked attributes in the map (use Ctrl+H for replace)	“Finding information in a map” on page 72
Ctrl+H	Display Replace dialog to find and replace text in customized Java Code and comments of transformation rules.	“Finding and replacing text” on page 73
Ctrl+I	Open a map definition file	“Steps for opening a map from a file” on page 57
Ctrl+M	View a map document	“Viewing a map document” on page 64
Ctrl+N	Display the New Map wizard to create a new map	“Creating a map: Basic steps” on page 30
Ctrl+O	Open a map definition from the project in System Manager	“Steps for opening a map from a project in System Manager” on page 56
Ctrl+P	Print the map definition	“Printing a map” on page 74
Ctrl+S	In Map Designer main window—Save the current map definition to the project in System Manager	“Saving a map to a project” on page 49

Table 12. Keyboard shortcuts for Map Designer (continued)

Keyboard shortcut	Description	For more information
Ctrl+U	Display the Preferences dialog to set Map Designer preferences	"Assigning preferences" on page 21
Ctrl+Alt+F	Save the current map definition to a map definition file with a different name (Save As)	"Saving a map to a file" on page 50
Ctrl+Alt+S	Save the current map definition to the project in System Manager with a different name (Save As)	"Saving a map to a project" on page 49
Ctrl+Shift+P	Display the Print Setup dialog to specify information for printing the map definition	"Printing a map" on page 74
Ctrl+Enter	Display the Map Properties dialog, from which you can set general and business object properties for the map	"Specifying map property information" on page 58
F7	Compile the current map	"Compiling a map" on page 84
Alt+F4	Close the current map	"Closing a map" on page 57
Del	Delete the currently selected entity	N/A
F1	Display context-sensitive help for the current dialog or window	N/A
Ctrl+F7	Compile all or a subset of maps defined in System Manager	"Compiling a set of maps" on page 86
F8	During a test run, continue a paused map by executing until the end of the map or another active breakpoint	"Steps for processing breakpoints" on page 93
F9	Toggle the state of a breakpoint for a transformation rule	"Setting breakpoints" on page 90
F10	During a test run, continue a paused map by executing the next single step	"Steps for processing breakpoints" on page 93

## Creating a map: Basic steps

Table 13 provides an overview of the subtasks for creating a new map.

Table 13. Subtasks for creating a new map

Subtask	Associated procedure (see . . . )
1. Creating a new map file with the New Map wizard, specifying the project, the source and destination business objects, and the name for the new map.	"Steps for creating the map definition" on page 31.
2. Setting the verb for each destination business object. In most cases, destination business objects have the same verb as source business objects. You can also set the value of the verb always to be a specific value.	"Setting the destination business object verb" on page 36.
3. Specifying the transformation steps for each destination attribute that you want to map. How you do this depends on what kind of transformation is required.	"Specifying standard attribute transformations" on page 36.
4. Specifying the comment for the destination attribute. Although this information is optional, it greatly improves readability of the map information in Map Designer.	"Setting comments in the comment field of the attribute" on page 52.
5. Saving the map.	"Saving maps" on page 49.
6. Checking completion, validating, and compiling the map.	"Checking completion" on page 51, "Validating a map" on page 84, and "Compiling a map" on page 84
7. Testing and debugging the map.	"Testing maps" on page 87



## Steps for creating the map definition

Map Designer provides a New Map wizard to assist you in creating a map definition. Perform the following steps to create a map definition using the New Map wizard:

1. Start the New Map wizard in one of the following ways:
  - From the File menu, select New to create a new map.
  - Use the keyboard shortcut of Ctrl+N.
  - In the Standard toolbar, click the New Map button.

**Result:** Map Designer displays the first window of the New Map wizard.

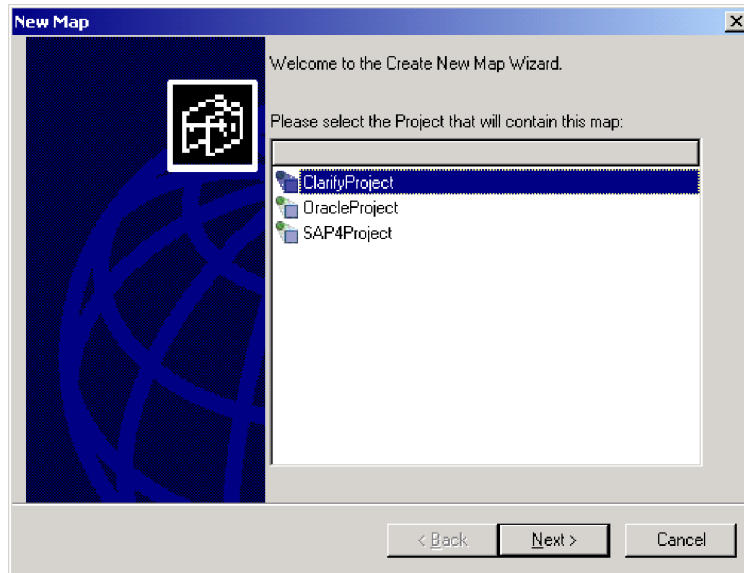


Figure 10. Welcome window of New Map wizard

2. From the list box, select the name of the project for which you want to create the map.
3. Select the business object you will use as the source business object for the map. You can select one or more source business objects by clicking in the Use

column of each desired business object. Then click Next to continue.

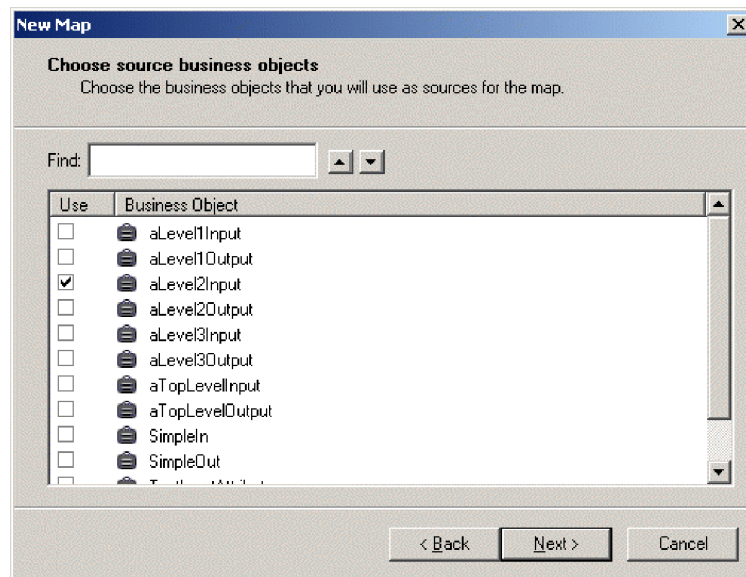


Figure 11. Selecting source business objects

**Tip:** To locate a particular business object, enter its name in the Find field. The up and down arrows scroll through the business object list. Click Next to continue.

The New Map wizard does *not* require that you specify the source business object. You can click Next without selecting the source business object to postpone specifying this business object definition. You can specify it at a later time in the map workspace of the Diagram tab. For more information, see "Creating the source and destination business objects" on page 34.

**Note:** If you add or delete a business object from System Manager, it dynamically updates the list of business object definitions.

4. Select the business object type you will use as the destination business object for the map. You can select one or more destination business objects by clicking

in the Use column of each desired business object. Then click Next to continue.

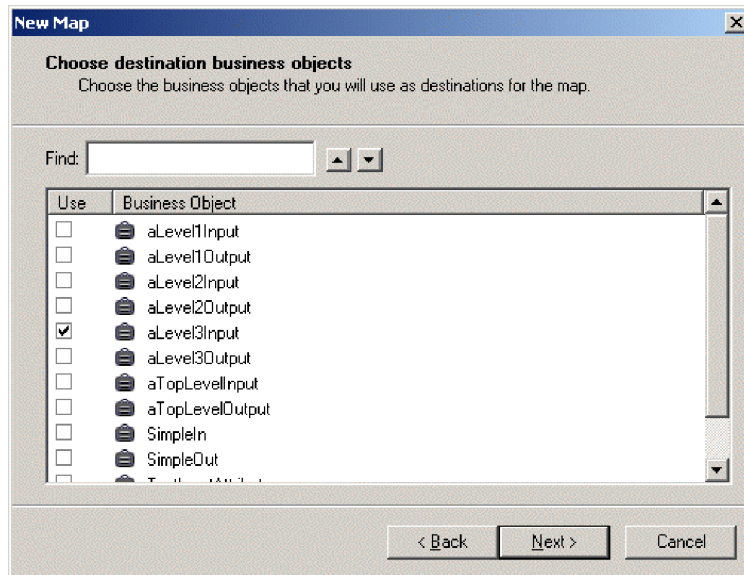


Figure 12. Selecting destination business objects

**Tip:** To locate a particular business object, enter its name in the Find field. The up and down arrows scroll through the business object list. Click Next to continue.

The New Map wizard does not require that you specify the destination business object. You can click Next without selecting the destination business object to postpone specifying this business object definition. You can specify it at a later time in the map workspace of the Diagram tab. For more information, see “Creating the source and destination business objects” on page 34.

**Note:** If you add or delete a business object from System Manager, it dynamically updates the list of business object definitions.

5. Specify the name to associate with the map.

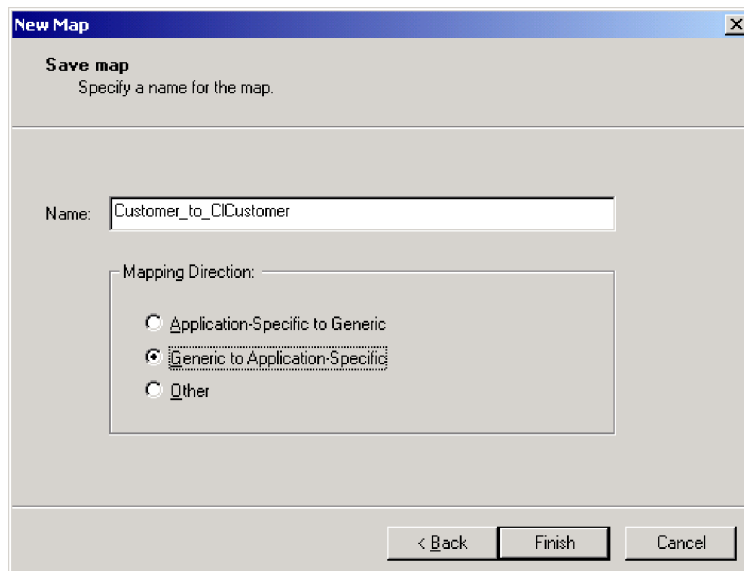


Figure 13. Saving new map

**Rule:** A map name must be less than or equal to 76 alphanumeric characters and underscores (\_). It *cannot* contain spaces or certain punctuation symbols, such as a period, a left brace(), a right brace (]), a single quotation mark, or a double quotation mark.

The New Map wizard does *not* require that you specify the map name. You can click Finish without entering the map name to postpone naming this map definition. When you save the map, Map Designer prompts you with the Save Map As dialog for you to specify the required map name. For more information, see “Saving a map to a project” on page 49.

Specify whether the map is an inbound or outbound map. This map role is needed for automatically generating relationship codes.

6. Click Finish to save the new map definition with the specified source and destination business objects.

**Result:** Map Designer displays the new map’s information in its Diagram tab.

## Creating the source and destination business objects

If you do not specify the map’s source and destination business objects from the New Map wizard, you can specify them from the Add Business Object dialog or the Diagram tab in the business object browser.

### Steps for specifying business objects from the Add Business Object dialog

Perform the following steps to add a source or destination business object to a map from the General tab of the Add Business Object dialog.

1. Display the Add Business Object dialog in one of the following ways:
  - From the Edit menu of Map Designer, select Add Business Object.
  - In the Designer toolbar, click the Add Business Object button.
  - From the Table tab, right-click in the empty area of the business objects pane and select Add Business Object from the Context menu.
  - From the Diagram tab, right-click in the map workspace and select Add Business Object from the Context menu.
2. To specify a source business object:
  - Click the business object in the business object list.
  - Click the Add to Source button.

**Tip:** To locate a particular business object, enter its name in the Find field. The up and down arrows scroll through the business object list.

3. To specify a destination business object:
  - Click the business object in the business object list.
  - Click the Add to Destination button.

**Tip:** To locate a particular business object, enter its name in the Find field. The up and down arrows scroll through the business object list.

4. To close the dialog, click Done.

### Steps for specifying business objects from the Diagram tab in the business object browser

From the Diagram tab, you can add a source or destination business object to a map. Perform the following steps to do this:

1. Drag the source business object from the business object browser to the left side of the map workspace. The business object displays and its title starts with Src.
2. Drag the destination business object from the business object browser to the right side of the map workspace. The business object displays and its title starts with Dest.

**Note:** A dotted-line boundary divides the left and right halves of the workspace and identifies the source and destination portions of the map workspace. Be sure to carefully drop objects in the appropriate place.

Figure 14 shows the source and destination business objects in the map workspace.

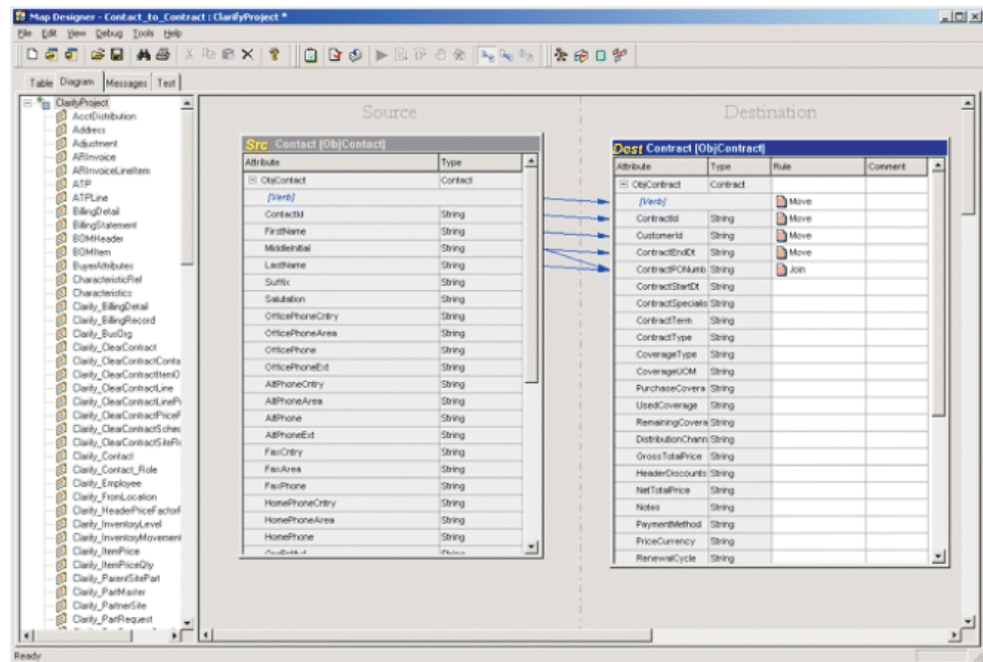


Figure 14. Defining Source and Destination business objects

**Tip:** Alternatively, you can create the source and destination business objects by right-clicking the business object in the business object browser; selecting Copy from the Context menu; then right-clicking in the map workspace and selecting Paste As Input Object or Paste As Output Object.

Map Designer creates a window, called a *business object window*, for the source and destination objects. The title bar of this window displays the business object instance name. For help interpreting the title bar of the business object window, see “Using generated business object variables and attributes” on page 169.. The business object window for the source business object contains columns for the name and data type of each source attribute. The business object window for the destination business object contains columns for the name, data type, transformation rule (which identifies the transformation step), and an optional comment.

**Guideline:** If you make a mistake by dragging the wrong business object or making it an output object instead of input, you can delete the object from the map workspace and try again. To delete a business object from the map workspace, you can either:

- Select the business object to delete and from the Edit menu select Delete Current Selection (or press the Del key).
- Right-click the title bar of the business object's window and select Delete from the Context menu.

## Setting the destination business object verb

The verb indicates how the system should process the business object's data. When a map executes, the system needs to know what verb to assign to each destination business object it creates.

If a map has only one source business object and one destination business object, the verb for the destination business object is usually the same as the verb for the source business object.

In this case, you need to copy the verb from the source business object to the destination business object (see Figure 14 on page 35), by defining a *Move* transformation rule with the source attribute as the source business object's verb and the destination attribute as the destination business object's verb. For more information, see "Copying a source attribute to a destination attribute" on page 38.

**Tip:** You can also drag-and-drop the verb from the source business object to the destination business object to define the value of the verb.

If a map has a destination business object with a verb that is not found in the source business object, you need to set the verb to a constant value, by defining a *Set Value* transformation rule with the destination attribute as the destination business object's verb. In the Set Value dialog box, enter the constant verb value. For more information, see "Specifying a value for an attribute" on page 37.

Maps sometimes have more than one source or destination business object, and these objects can have several child business objects. In these cases, you must consider carefully which verb to assign to each destination business object. Some destination business objects might require some custom logic to set the verb based on the verbs of one or more source business objects.

## Specifying standard attribute transformations

You can specify several standard attribute transformations interactively in Map Designer while writing little or no Java code. Table 14 shows the standard transformations that you can specify in Map Designer.

Table 14. Common attribute transformations

Name	Transformation step	Purpose
Set Value	"Specifying a value for an attribute" on page 37	For an attribute in the destination business object that is not found in the source business object but is required in the destination application
Move	"Copying a source attribute to a destination attribute" on page 38	For an attribute that is the same in both the source and destination business objects
Join	"Joining attributes" on page 39	For an attribute in the destination business object that is a combination of several attributes in the source business object



Table 14. Common attribute transformations (continued)

Name	Transformation step	Purpose
Split	"Splitting attributes" on page 41	For an attribute in the destination business object that is either: <ul style="list-style-type: none"> <li>• Only one part of an attribute in the source business object</li> <li>• Made up of several fields, but with different delimiters from those in the source business object</li> </ul>
Submap	"Transforming with a submap" on page 43	For attributes in the source and destination business objects that contain child business objects
Cross-Reference	"Cross-referencing identity relationships" on page 47	For maintaining the identity relationships for the business objects
Custom	"Creating a Custom transformation" on page 47	For an attribute that requires transformations not provided by the automatically generated transformations

For information on additional transformations you can perform, see "More attribute transformation methods" on page 173.

In the Diagram tab, you can select which attributes display in the business object windows with the options of the View > Diagram menu. You can choose to display all attributes, only linked (mapped) attributes, or only unlinked (unmapped) attributes.

**Tip:** Attributes appear in the same order that they appear in the business object definition. To locate a particular attribute in a long list of attributes, select Find from the Edit menu (or use the keyboard shortcut of Ctrl+F). For more information, see "Finding information in a map" on page 72..

### Specifying a value for an attribute

Some destination attribute values do not depend on a source attribute and can be filled in with a constant value. This is especially true if the destination business object contains many attributes that are not found in the source business object but are required in the destination application. Some examples of default values for attributes are CustomerStatus = "active" or AddressType = "business".

This type of transformation is called a *Set Value* transformation. You set the value of a destination attribute with the Set Value dialog, shown in Figure 15.

**Steps for specifying a Set Value transformation:** Perform the following steps to specify a Set Value transformation:

1. Display the Set Value dialog in one of the following ways:
  - From the Table tab, perform the following steps:
    - a. Select the destination attribute whose value you want to set.
    - b. Click Set Value from the list in the Transformation Rule column.
  - From the Diagram tab, perform the following steps:
    - a. Select the destination attribute whose value you want to set.
    - b. Click Set Value from the list in the Rule column of the destination business object.

- If a Set Value transformation is already defined, you can display the Set Value dialog to reconfigure the transformation, including modifying its transformation code in either of the following ways:
  - Double-click the corresponding cell of the transformation rule column.
  - Click the Set Value bitmap icon contained in the transformation rule column.

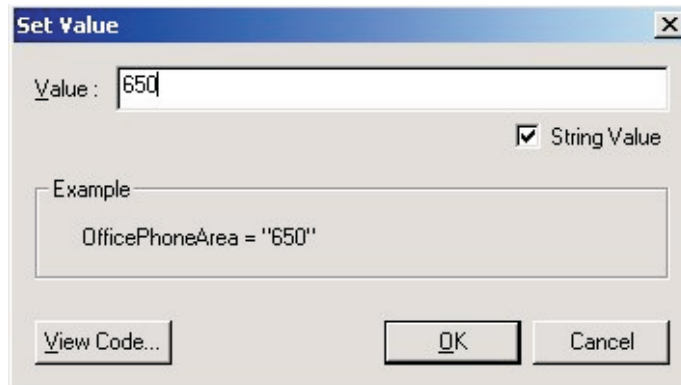


Figure 15. Set Value dialog

2. Through the Set Value dialog, you set the constant value to assign to the destination attribute. The Set Value dialog provides the following functionality:
  - To specify the constant value, enter it in the Value field. For numeric values, simply enter the number and make sure that the String Value check box is not selected. For string values, enter the string value in the Value field and select the String Value check box.

**Note:** The Set Value dialog uses the Examples area to show how the resulting destination attribute will look.

- To modify the value you have entered, click in the Value field and edit as appropriate.
- To customize the generated code, click the View Code push button.

**Result:** Map Designer opens Activity Editor in Java view, containing a sample of the transformation code in read-only mode for the destination attribute. To make changes to the transformation code, click Edit Code in Activity Editor. For more information, see “Overview of Activity Editor” on page 103.

**Note:** When you save the changes in Activity Editor, they are communicated to Map Designer. When you save the map, they are saved too.

- To confirm the transformation setting, click OK.

### Copying a source attribute to a destination attribute

The simplest kind of transformation step is a copy of one source attribute into a corresponding destination attribute. This type of transformation is called a *Move* transformation.

**Steps for specifying a Move transformation:** Perform the steps from one of these map tabs to specify a Move transformation:

- From the Table tab:



1. Select the source attribute.
  2. Select the destination attribute.
  3. Click Move from the list in the Transformation Rule column.
- From the Diagram tab:
    1. Select the source attribute.
    2. Use Ctrl+Drag to move to the destination attribute; that is, hold down the Ctrl key and drag the attribute onto the destination attribute in the destination business object window. Continue to hold down the Ctrl key until after you release the mouse button; otherwise, the operation does not succeed.

**Result:** Map Designer creates a blue arrow from the source to the destination object. If the transformation involves a single source attribute that is *not* a child business object, Map Designer assumes that the transformation is a Move and automatically assigns Move to the Rule column of the destination attribute.

**Tip:** You can customize the key sequence used to initiate a Move transformation in the Diagram tab from the Key Mapping tab of the Preferences dialog. For more information, see “Specifying Key Mapping” on page 23.

**Result:** Map Designer generates the code to copy the value of the source attribute to the destination attribute. If the source and destination attributes are of different data types, Map Designer determines whether a type conversion is possible, and if so, generates the code to convert the source type to the destination type. If a type conversion is *not* possible, or might result in data loss, Map Designer displays a dialog box for you to confirm or cancel the operation.

If you want to see a sample of the generated code for the Move transformation, in the Context menu of the rule column, select View Source.

## Joining attributes

You can concatenate, or join, the values from more than one source attribute into a single destination attribute. This type of transformation is called a *Join* transformation. For instance, the source business object might store the area code, telephone number, and extension in separate attributes, while the destination business object stores these values together in one attribute.

In addition to joining the attributes, you can reorder them and insert delimiters, parentheses, or other characters. For instance, when joining separate area code and telephone number attributes into a single attribute, you might want to insert parentheses around the area code.

**Tip:** The attributes you want to join can sometimes be located in more than one source business object, such as in a parent business object and one of its child business objects. You can also join an attribute with a variable you have defined. (To learn about defining variables, see “Creating temporary variables” on page 171.)

You join multiple source attributes into one destination attribute with the Join dialog, shown in Figure 16.

**Steps for specifying a Join transformation:** Perform the following steps to specify a Join transformation:

1. Display the Join dialog in one of the following ways:
  - From the Table tab:

- a. Select the source attributes to join.
 

**Tip:** You can click Multiple Attributes in the combo box to display the Multiple Attributes dialog. In this dialog, you can select multiple source attributes. To locate a particular business object, enter its name in the Find field. The up and down arrows scroll through the business object list. Once you have selected the source attributes, click OK to close the dialog.
  - b. Select the single destination attribute.
  - c. Click Join from the list in the Transformation Rule column.
- From the Diagram tab:
    - a. Select two or more source attributes.
    - b. Use Ctrl+Drag to move to the destination attribute; that is, hold down the Ctrl key and drag the selected source attributes to the destination attribute. Continue to hold down the Ctrl key until after you release the mouse button; otherwise, the operation does not succeed.
 

**Result:** If the transformation involves more than one source attribute, Map Designer assumes that the transformation is a Join. It automatically assigns Join to the Rule column of the destination attribute and displays the Join dialog.

**Tip:** You can customize the key sequence used to initiate a Join transformation in the Diagram tab from the Key Mapping tab of the Preferences dialog. For more information, see “Specifying Key Mapping” on page 23.
  - If a Join transformation is already defined, you can use the Join dialog to reconfigure the transformation, including modifying its transformation code, in either of the following ways:
    - Double-click the corresponding cell of the transformation rule column.
    - Click the Join bitmap icon contained in the transformation rule column.

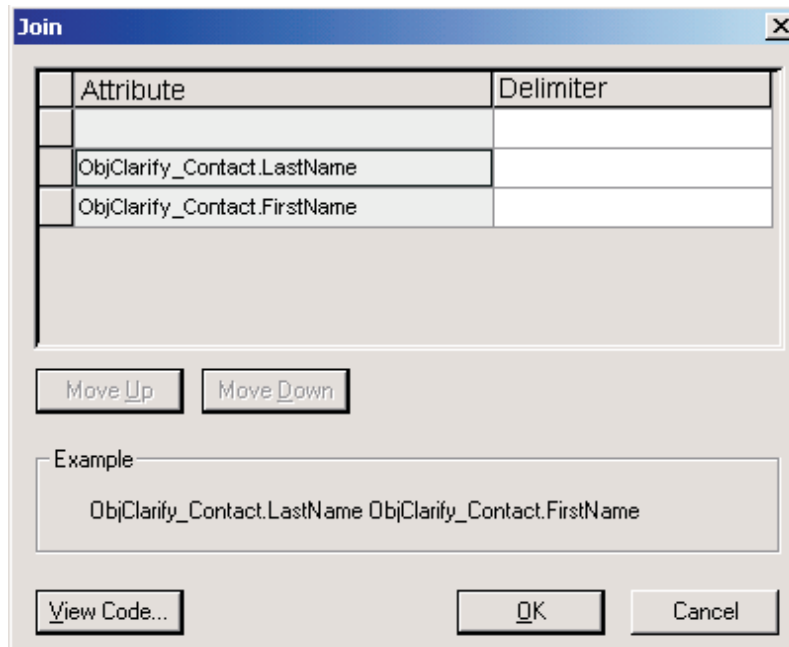


Figure 16. Join dialog

2. Through the Join dialog, you build an expression to concatenate the source attributes by adding delimiters, grouping with parentheses, and reordering the attributes if necessary. The Join dialog provides the following functionality:

- To insert a delimiter or parenthesis, enter it in the Delimiter field associated with the attribute. Do *not* put quotation marks around delimiters. The delimiter you enter is appended to the associated attribute. For leading delimiters, enter the delimiters in the Delimiters field of the initial blank line.

**Note:** The Join dialog uses the Examples area to show how the resulting string will look after the join.

- To modify a delimiter or parenthesis you have entered, click in the Delimiter field and edit as appropriate.
- To reorder a delimiter or the attributes, click the left-most column to select the row, then click Move Up or Move Down to move the whole row up or down.
- To customize the generated code, click the View Code push button.

**Result:** Map Designer opens Activity Editor in Java view, containing a sample of the transformation code in read-only mode for the destination attribute. To make changes to the transformation code, click Edit Code in Activity Editor. For more information, see “Overview of Activity Editor” on page 103.

**Note:** When you save the changes in Activity Editor, they are communicated to Map Designer. When you save the map, they are saved too.

- To confirm the transformation setting, click OK.

**Result:** Map Designer generates the code to join the source attributes. If any source attribute is of a different data type from the destination attribute, Map Designer makes the necessary calls to methods in the `DtpDataConversion` class to convert the types.

## Splitting attributes

To split a source attribute into two or more destination attributes, you specify the transformation for each destination attribute separately. This type of transformation is called a *Split* transformation. For instance, to split a source attribute, such as `phone_number`, into three separate destination attributes, such as `area_code`, `tel_number`, and `extension`, you specify the transformations for `area_code`, `tel_number`, and `extension` separately.

You split a source attribute into multiple destination attributes with the Split dialog, shown in Figure 17.

**Steps for specifying a Split transformation:** Perform the following steps to specify a Split transformation:

1. Display the Split dialog in one of the following ways:
  - From the Table tab, perform the following steps:
    - a. Select the single source attribute to split.
    - b. Select one of the desired destination attributes.
    - c. Click Split from the list in the Transformation Rule column.
    - d. Repeat these steps for each destination attribute that receives a segment of the source attribute.
  - From the Diagram tab, perform the following steps:
    - a. Select the single source attribute to split.

- b. Use Alt+Drag to move to one of the destination attributes; that is, hold down the Alt key and drag the source attribute to one of the destination attributes.

**Result:** If the transformation involves more than one destination attribute, Map Designer assumes that the transformation is a Split. It automatically assigns Split to the Rule column of the destination attribute and displays the Split dialog.

- c. Repeat these steps for each destination attribute that receives a segment of the source attribute.

**Tip:** You can customize the key sequence used to initiate a Split transformation in the Diagram tab from the Key Mapping tab of the Preferences dialog. For more information, see “Specifying Key Mapping” on page 23.

- If a Split transformation is already defined, you can use the Split dialog to reconfigure the transformation, including modifying its transformation code, in either of the following ways:
  - Double-click the corresponding cell of the transformation rule column.
  - Click the Split bitmap icon contained in the transformation rule column.

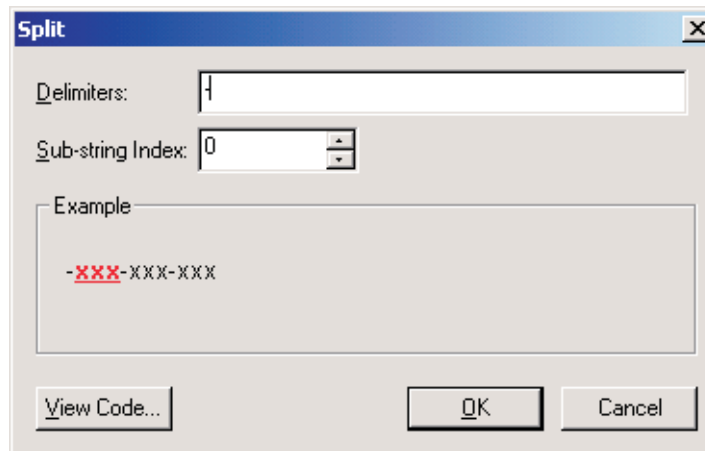


Figure 17. Split dialog

- 2. Through the Split dialog, you split an expression into segments that are separated by a delimiter. Each segment is identified with an index number, with the first segment having an index number of zero (0). The Split dialog provides the following functionality:
  - To identify the delimiter by which to parse the source attribute, enter it in the Delimiter field. Do *not* put quotation marks around delimiters. You can specify one or more delimiters in this field. The transformation uses each of the specified delimiters to parse the string into segments. For example, to split LastName,FirstName, specify “,” as the delimiter, LastName as segment 0 (the first segment) and FirstName as segment 1 (the second segment).

**Note:** The Split dialog uses the Examples area to show how the source attribute string looks and to indicate which segment is currently being accessed. The accessed segment displays in bold and red.

- To modify a delimiter or parenthesis you have entered, click in the Delimiter field and edit as appropriate.

- To identify the segment of the source attribute that is copied to the destination attribute, enter its index number in the Sub-string Index field.
- To customize the generated code, click the View Code push button.

**Result:** Map Designer opens Activity Editor in Java view, containing a sample of the transformation code in read-only mode for the destination attribute. To make changes to the transformation code, click Edit Code in Activity Editor. For more information, see “Overview of Activity Editor” on page 103.

**Note:** When you save the changes in Activity Editor, they are communicated to Map Designer. When you save the map, they are saved too.

- To confirm the transformation setting, click OK.

**Result:** Map Designer generates the transformation code for the destination attribute. The generated code uses methods from the `DtpSplitString()` class to parse the source attribute into segments.

### Transforming with a submap

A *submap* is a map that is called from within another map, called the *main map*. This section provides the following information about submaps:

- “Uses for submaps”
- “Steps for specifying a Submap transformation” on page 45

**Uses for submaps:** You can call a submap to obtain a value for any destination attribute, but submaps are most commonly used for the following:

- To modularize a map
- To specify transformations between child business objects

*Improving map modularity:* Using submaps can improve the modularity of your maps by isolating common transformations that can be reused in more than one map. For example, a Customer business object might have an Address child business object that is also a child of an Order business object. If you create a submap for the Address business object, you can reuse the submap in both the Customer and Order business object maps.

Figure 18 illustrates how a submap, `MyAddrToGenAddr`, can be reused by two different maps.

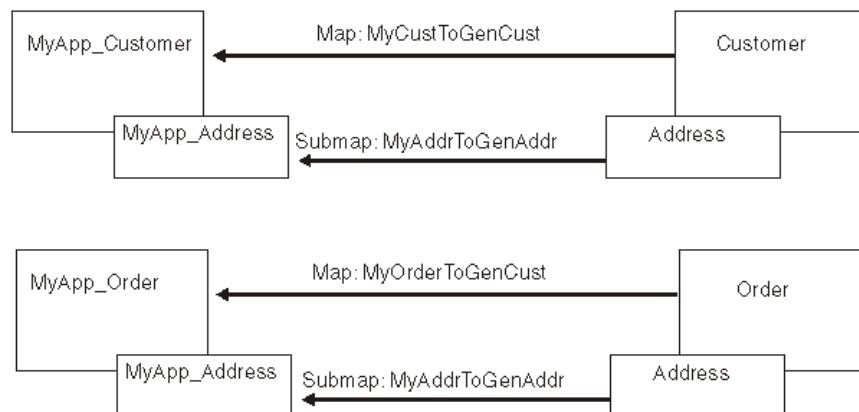


Figure 18. Using submaps for modularity

*Transforming child business objects:* When the source and destination attributes contain multiple-cardinality child business objects, it is useful to use a submap to specify their transformations. Typical examples of multiple-cardinality child business objects are the multiple addresses of a customer or the multiple line items in an order.

In the simplest case, you transform each source child business object into a single destination child business object, in a one-to-one relationship. Figure 19 illustrates the use of submaps for an Employee business object and its child business array that contains instances of EmployeeAddress.

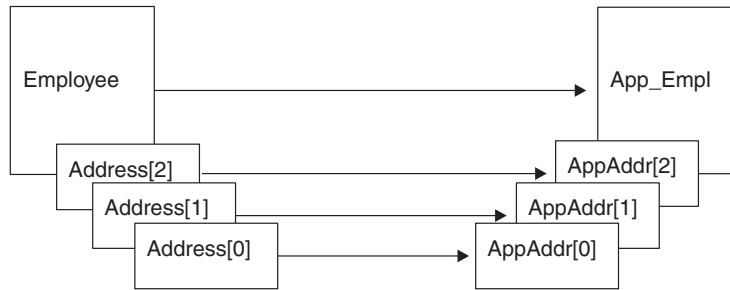


Figure 19. One-to-one transformation of child business object arrays

A submap can be associated with a conditional statement that governs whether it executes. For example, consider Figure 20: the Order business object has an OrderLine attribute that contains a multiple-cardinality child business object, OrderLine. The OrderLine business object has a DeliverySchedule attribute that contains a multiple-cardinality child business object, DelSched.

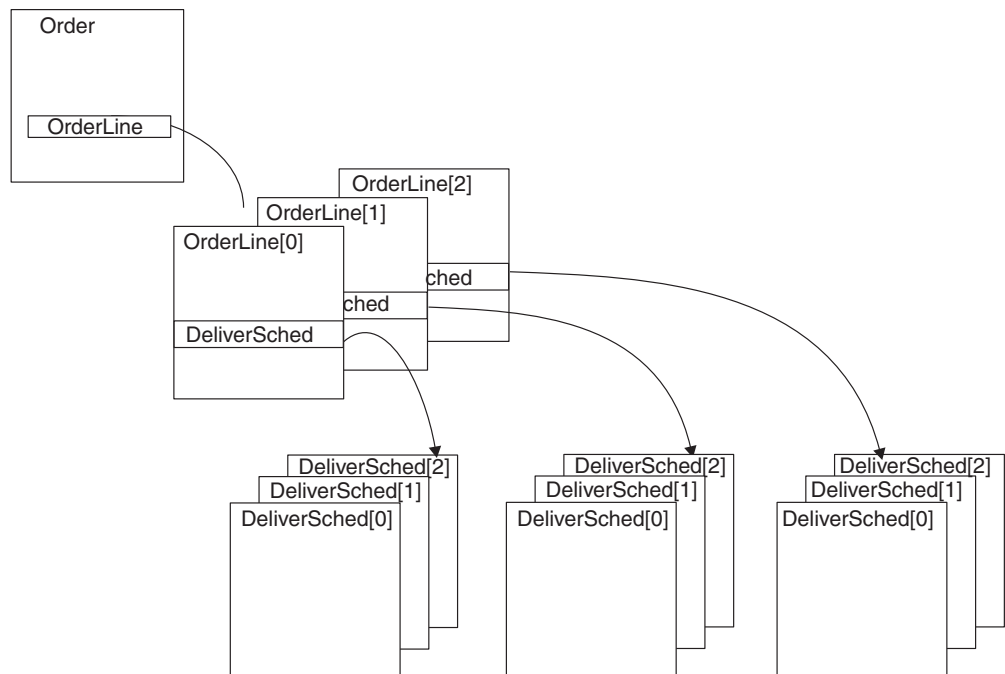


Figure 20. Source business object with multiple-cardinality child business object

Some conditions that can be written in the map for Order can:

- Execute the submap that transforms the OrderLine attribute in Order only if a different attribute in Order has a particular value.
- Execute the submap that transforms the DeliverSched attribute in OrderLine only if a different attribute in OrderLine has a particular value.
- Execute the submap that transforms the DeliverSched attribute in OrderLine only if an attribute in Order has a particular value.

**Steps for specifying a Submap transformation:** Perform the following steps to specify a Submap transformation:

1. Create the map that you want to use as a submap.  
**Recommendation:** You do this in the same way that you create and save any other map. IBM naming conventions suggest that submap names begin with the string “Sub\_”.
2. Save the submap to the project in System Manager and compile the submap.
3. Specify the Submap transformation on the attribute in the parent business object that needs to call the submap. This source attribute contains a child business object that is mapped to a destination attribute that contains a child business object.

You specify that a submap needs to be called with the Submap dialog, shown in Figure 21.. Display the Submap dialog in one of the following ways:

- From the Table tab, perform the following steps:
  - a. In the parent map, select a source attribute (which is a child business object).
  - b. Select the desired destination attribute (which is also a child business object).
  - c. Click Submap from the list in the Transformation Rule column.
  - d. Repeat these steps for each source attribute that is a source business object for the submap and each destination attribute that is a destination business object for this submap.
- From the Diagram tab, perform the following steps:
  - a. In the parent map, select the source attribute (which is a child business object).
  - b. Use Ctrl+Drag to move to the destination attribute; that is, hold down the Ctrl key and drag the source attribute onto the destination attribute. Continue to hold down the Ctrl key until after you release the mouse button; otherwise, the operation does not succeed.

If the transformation involves a source attribute that is a child business object, Map Designer assumes that the transformation is a Submap. It automatically assigns Submap to the Rule column of the destination attribute and displays the Submap dialog.

**Tip:** You can customize the key sequence used to initiate a Submap transformation in the Diagram tab from the Key Mapping tab of the Preferences dialog. For more information, see “Specifying Key Mapping” on page 23.

- If a Submap transformation is already defined, you can use the Submap dialog to reconfigure the transformation, including modifying its transformation code, in either of the following ways:
  - Double-click the corresponding cell of the transformation rule column.



- Click the Submap bitmap icon contained in the transformation rule column.

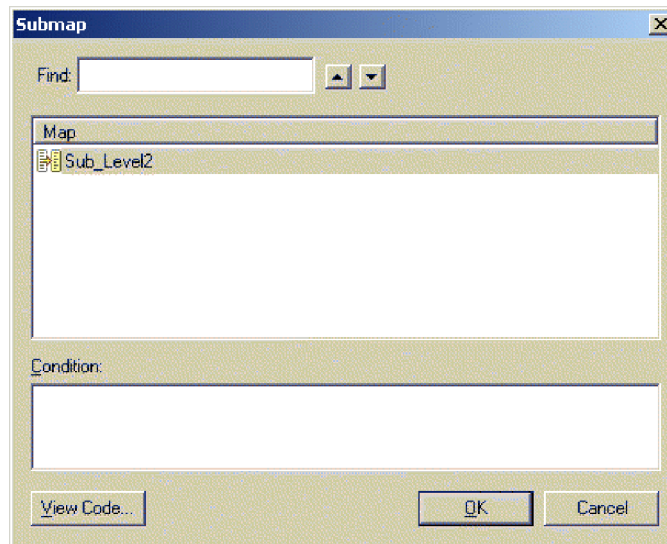


Figure 21. Submap dialog

- Through the Submap dialog, you specify the name of the submap to call. The Submap dialog provides the following functionality:
  - To identify the submap to call, select its name from the list in the Map area. The map list displays maps that meet the following condition: The submap has the same business object definitions for its source and destination business objects as the source and destination attribute you have selected.
 

**Tip:** To locate a particular submap, enter its name in the Find field. The up and down arrows scroll through the business object list.
  - To specify a condition for the submap, enter it in the Condition area of the Submap dialog. You can enter the condition now or simply dismiss the dialog and enter the condition in the destination attribute's generated code.
  - To customize the generated code, click the View Code push button.
 

**Result:** Map Designer opens Activity Editor in Java view, containing a sample of the transformation code in read-only mode for the destination attribute. To make changes to the transformation code, click Edit Code in Activity Editor. For more information, see "Overview of Activity Editor" on page 103.

**Note:** When you save the changes in Activity Editor, they are communicated to Map Designer. When you save the map, they are saved too.
  - To confirm the transformation setting, click OK.

**Result:** Map Designer generates the Java code to call the specified submap. It automatically creates a call to the `runMap()` method to call the submap.

**Note:** In any attribute's code, you can use Expression Builder to insert a map execution call. For more information, see "Using Expression Builder to call a submap" on page 200.

**Reminder:** If you have saved a transformation rule using *custom Java code*, you will need to manage any object dependencies that are used in the custom



Java code *manually*. For example, if the custom Java code calls a submap, then you will need to manually deploy the submap to the server.

## Cross-referencing identity relationships

In some cases, the source attribute may need to reference a relationship table to find out what value to set in the destination attribute. This can be done using a *Cross-Reference* transformation.

**Steps for specifying a Cross-Reference transformation:** Perform the following steps to specify a Cross-Reference transformation:

1. Select the source and destination attributes in any of the ways previously described for other transformations. Both have to be business objects.
2. Select Cross-Reference in the corresponding transformation cell.

**Result:** The Cross-Reference dialog appears:

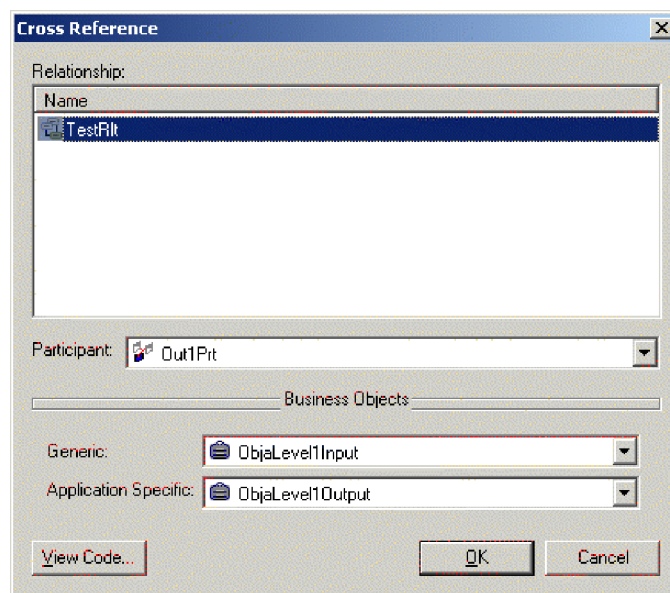


Figure 22. Cross-Reference dialog

3. In this dialog, select the relationship name from the list.

**Result:** The Participant combo box will be populated with all participants from the selected relationship. The Business Object combo box, by default, will be populated according to the mapping role defined in the map property. You can change the combo boxes.

## Creating a Custom transformation

In a *Custom* transformation, you use Activity Editor to customize the activity for the transformation graphically or to enter the Java code to transform the source attribute to the destination attribute.

**Tip:** If you want to use only one standard function block in a custom transformation, you can configure the function block in the Preferences dialog for direct use in Map Designer. For more information, see “Tip: Using function blocks directly in Map Designer” on page 112

**Steps for specifying a Custom transformation:** Perform the steps from one of these map tabs to define a Custom transformation:

- From the Table tab:
  1. Select the source attribute.
  2. Select the desired destination attribute.
  3. Click Custom from the list in the Transformation Rule column.
- From the Diagram tab:
  1. Select the source attribute.
  2. Select the desired destination attribute.
  3. Drag the source attribute onto the destination attribute in the destination business object window.
- If a custom transformation is already defined, you can modify its transformation code in either of the following ways:
  - Double-click the corresponding cell of the transformation rule column.
  - Click the Custom bitmap icon contained in the transformation rule column.

**Tip:** You can customize the key sequence used to initiate a Custom transformation from the Key Mapping tab of the Preferences dialog. For more information, see “Specifying Key Mapping” on page 23.

**Result:** Map Designer displays Activity Editor with a graphical view. For more information on Activity Editor, see “Overview of Activity Editor” on page 103.

**Reminder:** If you have saved a transformation rule using *custom Java code*, you will need to manage any object dependencies that are used in the custom Java code *manually*. For example, if the custom Java code calls a submap, then you will need to manually deploy the submap to the server.

Table 15 lists information in this guide that is useful in defining a custom transformation.

Table 15. Defining custom transformations

Information provided	For more information
How to use Activity Editor to customize transformation code	Chapter 5, “Customizing a map,” on page 103
How to create relationships for relationship attributes	For a general introduction to relationships, see Chapter 6, “Introduction to relationships,” on page 223.
1. Use Map Designer to create the map for the business objects that contain relationships.	Chapter 2, “Creating maps,” on page 15
2. Use Relationship Designer to define the relationship.	Chapter 7, “Creating relationship definitions,” on page 237
3. Return to Map Designer to code the relationship between the attributes.	Chapter 8, “Implementing relationships,” on page 257
More complex transformations you can perform:	“More attribute transformation methods” on page 173
Content-based logic	“Content-based logic” on page 174
Date formatting	“Date formatting” on page 179
String processing	“Using Expression Builder for string transformations” on page 182

**Note:** You can also customize an existing transformation by modifying the generated code from Activity Editor. If you modify code in auto-update mode, Activity Editor prompts for a confirmation. If you confirm the

change, Activity Editor saves the customized code. The label of the transformation icon in the Transformation Rule column of the Table or Diagram tab changes from displaying in black, normal text to displaying in blue, italic text. These blue icon labels help you distinguish between code that is in auto-update mode (generated by Map Designer) and code you have customized.

You can tell Activity Editor not to confirm by changing the setting in the Preferences dialog.

## Saving maps

To preserve the map definition for use at a later time, you must save the map. Before Map Designer saves a map, it first validates the map. For more information, see “Validating a map” on page 84.

Map Designer provides two ways to save the current map:

- “Saving a map to a project” on page 49
- “Saving a map to a file” on page 50

**Important:** For Map Designer to be able to save a map, a map must currently be open.

### Saving a map to a project

A map definition stores map information in a project in System Manager. This map definition contains the following information for a map:

- The general map information, which includes map properties
- The map design, which includes the transformation mappings
- The custom transformation code

To save a map to a project in System Manager, you can perform any of the actions shown in Table 16.

*Table 16. Saving a map to the project*

<b>If you want to . . .</b>	<b>Then . . .</b>
Save the map definition to the name of the currently open map.	Do one of the following: <ul style="list-style-type: none"><li>• Select To Project from the File &gt; Save submenu.</li><li>• Use the keyboard shortcut of Ctrl+S.</li><li>• In the Standard toolbar, click the Save Map to Project button).</li></ul>
Save the map definition to a name different from the currently open map.	Do one of the following: <ul style="list-style-type: none"><li>• Select To Project from the File &gt; Save As submenu.</li><li>• Use the keyboard shortcut of Ctrl+Alt+S.</li></ul>
	<b>Result:</b> Map Designer displays the Save Map As dialog in which you can specify the map name.

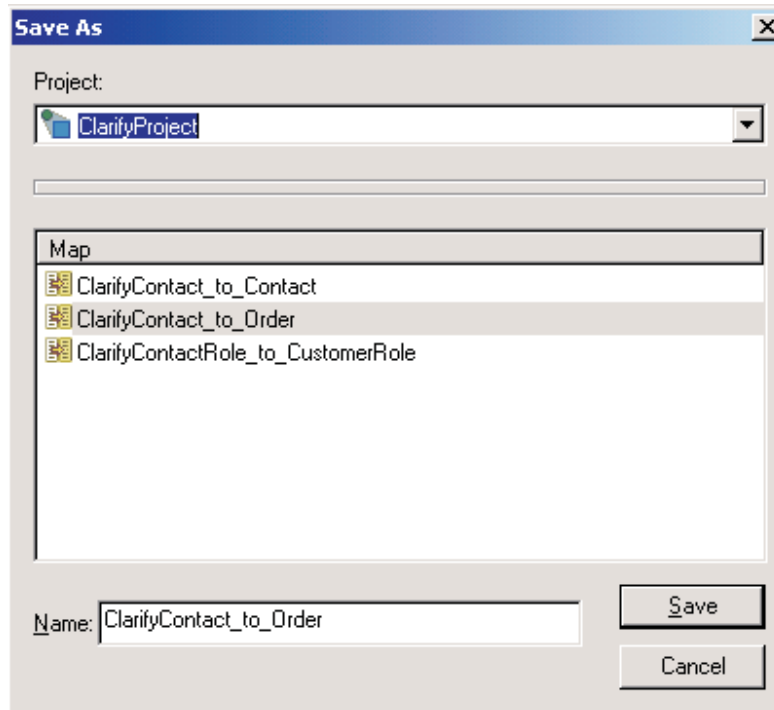


Figure 23. Save As dialog

When you save the map, Map Designer saves the map definition and map content to the project in System Manager. It saves the map content as XML data.

**Note:** You can specify whether Map Designer automatically saves a map to the project in System Manager before compiling the map with the option *Compile Map: save map before compile*. By default, this option is enabled. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

**Tip:** To rename an existing map, select *To Project* from the *File > Save As* submenu.

### Saving a map to a file

A map definition can be stored as text in an operating-system file, called a *map definition file*. A map definition file contains the complete map definition; that is, this file uses Extended Markup Language (XML) format to represent the following parts of a map definition:

- The general map information, which includes map properties
- The map content, which includes the transformation mappings in an uncompressed format

**Recommendation:** Map Designer creates a map definition file with a *.cwm* extension. You should follow a naming convention for your map definition files, such as using the file extension (*.cwm*) to distinguish them.

You import a map definition into Map Designer by opening an existing map definition file. For more information, see “Steps for opening a map from a file” on page 57.

You can save the currently open map to a map definition file in any of the ways shown in Table 17.

*Table 17. Saving a map to a map definition File*

<b>If you want to . . .</b>	<b>Then . . .</b>
Save the map to the name of the currently open map in the format: <i>MapName.cwm</i>  (where <i>MapName</i> is the name of the currently open map) <b>Note:</b> Map Designer will always open the File Save dialog if you do not open the currently opened map from file.	Do one of the following: <ul style="list-style-type: none"> <li>• Select To File from the File &gt; Save submenu.</li> <li>• Use the keyboard shortcut of Ctrl+E.</li> <li>• In the Standard toolbar, click the Save Map to File button (see Figure 23).</li> </ul>
Save the map to a specified map definition file. Map Designer displays a dialog box to allow you to select the file name.	Do one of the following: <ul style="list-style-type: none"> <li>• Select To File from the File &gt; Save As submenu.</li> <li>• Use the keyboard shortcut of Ctrl+Alt+F.</li> </ul>

**Note:** When you select the To File option from the File > Save or File > Save As menus, Map Designer displays a dialog box to allow you to select the file name. This file name identifies the file. It is not necessarily the name of the map.

**Example:** You can save MapA in a file named fileA.cwm. This fileA file contains the map definition for MapA. When Map Designer opens the fileA map definition file, it displays the MapA map definition.

**Tip:** Exporting a map copies only the map.

## Checking completion

When you are mapping two large business objects, it is easy to overlook some required attributes. You can search for attributes that are not yet mapped to make sure that you have specified all desired transformations. Such attributes are called *unlinked attributes*.

Perform the following step to check completion:

- From the Edit menu Select Find; and click the Unlinked attributes option in the Find control pane.

**Result:** Map Designer displays a list of attributes for which there is no transformation code. For more information, see “Finding information in a map” on page 72.

**Note:** Once the code is completed, you must compile and test it. For information on compiling a map, see “Compiling a map” on page 84. For information on testing a map, see “Testing maps” on page 87..

---

## Mapping standards

This section provides the following procedural standards for maps:

- “Tips on mapping individual attributes”
- “Setting comments in the comment field of the attribute”

### Tips on mapping individual attributes

The following points provide a general approach to mapping individual attributes:

- If the attribute mapping does *not* include relationship management, start by copying the source attribute to the destination attribute (see “Copying a source attribute to a destination attribute” on page 38), then modify the generated code, as needed.
- If the attribute mapping requires a call to a method in the Mapping API, write the code without copying the attribute.
- If the destination attribute requires a default when the source attribute is null, copy the attribute and note that the generated code includes two if statements for checking the source attribute. You can either:
  - Provide the default in an else statement for both of the if statements.
  - Add another if statement at the beginning of the code that checks the source attribute for null and adds a default value. Place the rest of the code in the else statement.

**Important:** Do *not* map the ObjectEventId attribute. InterChange Server reserves the ObjectEventId for its own processing purposes. Any custom code that has ObjectEventId as destination attribute will not execute properly.

### Setting comments in the comment field of the attribute

Attribute comments can improve the readability of your map. However, Map Designer does *not* automatically generate a comment for an attribute. Table 18 provides some suggested standards for attribute comments based on the type of transformation associated with the destination attribute.

Table 18. Settings for the Attribute Comment

Situation	Setting for Attribute Comment
If the child business object is <i>not</i> mapped	=No mapping
Set Value transformation	=SET VALUE( <i>value</i> )
Move transformation	=MOVE
Join transformation	=JOIN( <i>srcAttr1</i> , <i>srcAttr2</i> , ...)
Split transformation	=SPLIT( <i>srcAttr[index]</i> )
For child business objects, when the mapping is done <i>without</i> calling a submap to indicate the object has to be expanded to see its attributes	=Mapping here
If the code to call the submap is generated	=SUBMAP( <i>mapName</i> )
If the attribute’s mapping contains Mapping API calls that implement relationships, such as:	=Relationship( <i>type</i> )
• retrieveInstances()	where <i>type</i> can be:
• retrieveParticipants()	• identity
• maintainSimpleIdentityRelationship()	• lookup
• maintainCompositeRelationship()	• custom
• All other methods in the IdentityRelationship class <i>except</i> foreignKeyLookup() and foreignKeyXref()	
If the attribute’s mapping contains foreignKeyLookup()	=foreignKeyLookup()

Table 18. Settings for the Attribute Comment (continued)

Situation	Setting for Attribute Comment
If the attribute's mapping contains <code>foreignKeyXref()</code>	<code>=foreignKeyXref()</code>
Custom transformation that is <i>not</i> one of those listed above (relationship or foreign key)	<code>=CUSTOM(summary)</code>
If the attribute's code does not contain anything except setting the verb	<code>=SET VERB</code>





---

## Chapter 3. Working with maps

This chapter describes some advanced features of Map Designer that you might use after creating maps.

The chapter covers the following tasks:

- “Opening and closing a map” on page 55
- “Specifying map property information” on page 58
- “Designing maps for bidirectional languages” on page 60
- “Using map documents” on page 60
- “Using map automation” on page 65
- “Finding information in a map” on page 72
- “Finding and replacing text” on page 73
- “Printing a map” on page 74
- “Deleting objects” on page 74
- “Using execution order” on page 77
- “Creating polymorphic maps” on page 78
- “Importing and exporting maps from InterChange Server” on page 79

---

### Opening and closing a map

Map Designer displays one map at a time within the tab window. This map is called the *current map* (sometimes called the “currently open map”). You can control which map is the current map with the following Map Designer procedures:

- “Opening a map”
- “Closing a map” on page 57

#### Opening a map

A map must be open in Map Designer before you can view its information in a Map tab or modify this information. When Map Designer opens a map, if the validate map when open preference is enabled, it first performs a set of validations on this map.

**Note:** You can specify whether Map Designer validates a map when it opens it, with the option Open Map: validate map when open. By default, this option is enabled.

If this preference is enabled when a map that uses big business objects (that is, thousands of attributes) is opened, Map Designer may take a long time to open the map. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

The validations that Map Designer performs on the map are as follows:

- Ensures that each business object definition that the map uses is defined in the project in System Manager.
- Ensures that every attribute in the map exists in the specified business object definition, as defined in the project in System Manager.

- Ensures that the type of each attribute in the map matches its type in the specified business object definition, as defined in the project in System Manager.
- Validates transformations:
  - Ensures execution order is correct; that is, that execution order is unique, positive, and consecutive.
  - Ensures that no attributes have cyclic dependencies on each other. If any cyclic transformations are found, Map Designer displays the cyclic rules in the output window.
  - Checks transformation information:
    - Move transformation—only one source attribute is involved.
    - Join transformation—more than one source attribute is involved.
    - Split transformation—only one source attribute is involved; split index is greater than or equal to zero; split delimiter is not empty.
    - Set Value transformation—no source attribute is involved; a value has been specified.
    - Submap transformation—at least one source attribute is involved; submap name is specified.
    - Cross-Reference transformation—only one source attribute is involved.

Map Designer provides the following ways to open a map:

- “Steps for opening a map from a project in System Manager” on page 56
- “Steps for opening a map from a file” on page 57

### Steps for opening a map from a project in System Manager

Perform the following steps to open a map from a project in System Manager:

1. Open the Open a Map from a Project dialog in one of the following ways:
  - Click File > Open > From Project.
  - Use the keyboard shortcut of Ctrl+0.
  - In the Standard toolbar, click the Open Map from Project button.

**Result:** Map Designer displays the Open Map dialog.

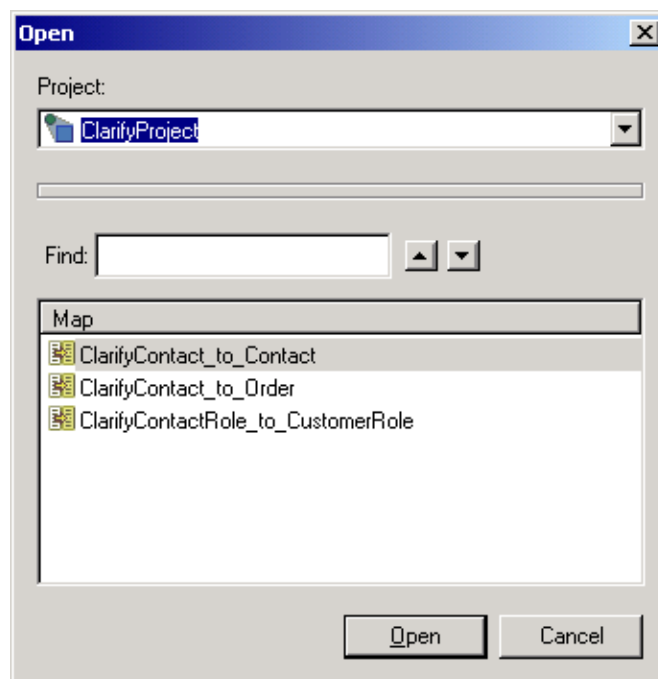


Figure 24. Open Map dialog

2. Select the project.
3. Select the map's name from the list of maps currently defined in the project in System Manager.  
**Tip:** To locate a particular map name, enter its name in the Find field. The up and down arrows scroll through the map list.
4. Click the Open button to open the map from the project.

### Steps for opening a map from a file

A map definition can be stored in XML format in an operating-system file called a *map definition file*. To create a map definition file, you save the map as a map design file (.cwm) in Map Designer. For more information, see "Saving a map to a file" on page 50.

When you open a map definition file, you open the map in Map Designer.

Perform the following steps to open a map definition file:

1. Open the Open a Map from a File dialog in one of the following ways:
  - Click File > Open > From File.
  - Use the keyboard shortcut of Ctrl+I.
  - In the Standard toolbar, click the Open Map from File button.

**Result:** The Open file with Map dialog box appears.

2. Select the map definition file you want to open. The file must be a .cwm file created by Map Designer.

**Result:** Map Designer opens the map definition file. The map information appears in the Map tabs.

**Important:** Opening the map in Map Designer does *not* automatically save the map to the project. To save this map to the project, continue to step 3..

3. Save the map to the project in System Manager. For more information, see "Saving a map to a project" on page 49..

**Rule:** You must save the map to the project in System Manager for it to be compiled. To compile the map, select Compile from the File menu. For more information, see "Testing maps" on page 87.

## Closing a map

Perform one of the following actions to close the current map, which is displaying in the tab window:

- Open a new map in any of the ways discussed in "Opening a map" on page 55.  
**Result:** Map Designer closes the current map before it opens a new one.
- From the File menu, select Close.  
**Result:** Map Designer closes the current map and clears the tab window. To make a new map current, you can either create a new map or open an existing map.
- Exit from Map Designer in one of the following ways:
  - From the File menu, select Exit.
  - Use the keyboard shortcut of Alt+F4.

**Result:** Map Designer automatically closes the current map before it exits.

**Note:** If you have changed the current map since it was last saved, Map Designer displays a confirmation box to confirm the map closure.

---

## Specifying map property information

Use the Map Properties dialog (see Figure 25) to display and specify property information for a map. To display the Map Properties dialog, perform any of the following actions:

- From the Edit menu, select Map Properties.
- Use the keyboard shortcut of Ctrl+Enter.
- In the map workspace of the Diagram tab, right-click and select Map Properties from the Context menu.

The Map Properties dialog provides the following tabs:

- General tab
- Business Objects tab

Figure 25 shows the General tab of the Map Properties dialog.

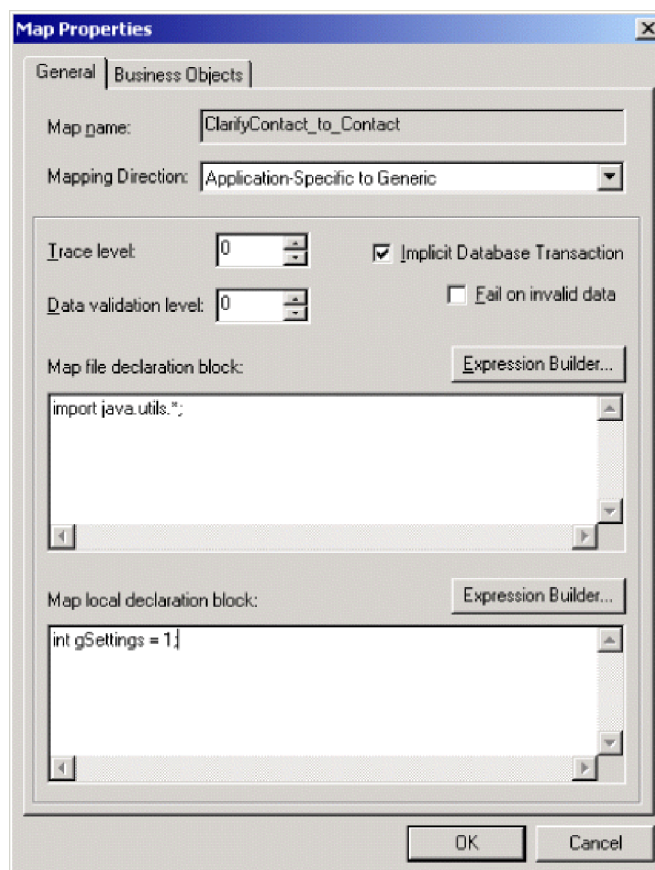


Figure 25. General tab of Map properties dialog

## Defining General Property information

The General tab of the Map Properties dialog displays the general property information shown in Table 19.

Table 19. General Map Property Information

General Map Property	Description	For more information
Map name	Identifies the map whose properties the dialog displays. This field is initialized when you create a new map and is not an editable field.	N/A
Mapping role	Identifies the purpose of the map. Possible values of mapping roles are: <ul style="list-style-type: none"> <li>• Application-specific to generic</li> <li>• Generic to application-specific</li> <li>• Other (for maps that do not have a specific mapping direction associated with them)</li> </ul> <p><b>Note:</b> For previously defined maps that do not have this property information, the combo box will be empty. This is permissible as long as you do not use any Relationship transformation rules. When you first create a Relationship transformation rule and this value is empty, Map Designer will prompt you for this value.</p>	
Run-time properties	Specifies the map properties (trace level, data validation level, implicit database transaction, and fail on invalid data) that apply to the map instance at run time. You can specify these properties here in the General tab of Map Designer's Map Properties dialog or from the Map Properties window that System Manager provides. The changes are made to the local file system. Deploying the map to the server will not update the run-time instance. <p><b>Note:</b> You can update these map properties dynamically from the server component management view by right-clicking on a map and selecting the properties from the Context menu. The changes will be automatically updated to the server.</p>	
Trace level	Sets the trace level for the map.	"Adding trace messages" on page 504
Data validation level	Allows you to check each operation in a map and log an error when data in the incoming business object cannot be transformed.	"Creating custom data validation levels" on page 187
Implicit Database transaction	Determines whether InterChange Server uses implicit transaction bracketing for transactions over its connections.	
Fail on invalid data	Determines whether map execution fails if data is invalid.	"Creating custom data validation levels" on page 187
Variable declarations	You can declare your own Java variables to use in your transformation code. For more information, see "Using variables" on page 169.	
Map file declaration block	Allows you to import Java packages (such as MapUtils) into a map for use within transformation code.	"Importing Java packages and other custom code" on page 164
Map local declaration block	Allows you to import custom Java code developed outside of Map Designer into a map for use within transformation code.	"Importing Java packages and other custom code" on page 164

## Defining business objects

The Business Objects tab of the Map Properties dialog displays information about the map's business objects. It lists the source and destination business objects as well as any temporary business object that might be defined. For more information, see "Steps for modifying business object variables" on page 170.

---

## Designing maps for bidirectional languages

The WebSphere Business Integration Server supports bidirectional languages. This support is in a standard Windows-type bidirectional format (logical left to right). With this support, data that flows into the InterChange Server environment that comes from one of the nine bidirectional language enabled connectors is guaranteed to be in uniform bidirectional language format (CWBF=ILYNN). However, data can be introduced into a map from sources other than enabled connectors, for instance, a component that does not support bidirectional languages that is exported through Web services, an adapter that does not support bidirectional languages or data imported from some external source where the bidirectional support is unknown. For more information see, "Using bidirectional functionality in Activity Editor" on page 162

Using non-bidirectional enabled sources can create bidirectional format inconsistencies that cause comparisons within a collaboration to return incorrect results. These types of errors can be avoided by:

- Only accepting input from sources that enforce the same bidirectional format as the WebSphere Business Integration Server such as the adapters that are already enabled with this support.
- Enabling the connectors to this collaboration to enforce the correct bidirectional format (see *Enabling connectors for bidirectional languages*) in the *Collaboration Development Guide*.
- Using the APIs in the CwBidiEngine class to transform all data into a consistent bidirectional format (see Chapter 12, "CwBidiEngine class," on page 361).

---

## Using map documents

You can create a *map document* to see all transformations in a single map or between two maps. While checking a map, you might want to view all of its transformations in a single operation, rather than opening and viewing each attribute separately. To do so, you can create a map document that contains all transformations. A map document provides you with an automated way to document native-map transformations.

This section provides the following information:

- A description of the two HTML files that make up a map document
- How to create a new map document
- How to view a map document
- How to print out a map document

### What is a map document?

A *map document* consists of two HTML files that describe all transformations of a map (or set of maps):

- A map-table file that describes the map transformations in a tabular format.  
The map-table file has the name *mapDoc*.HTM.

- A Java-code file that contains the code of the map transformations.  
The Java-code file has the name *mapDocJavaCode.HTM*.

In both these HTML files, *mapDoc* is the user-specified name of the map document.

The map document can include information for all attributes, only those attributes that have map transformations, or only those attributes that do not have map transformations (unlinked attributes). If you specify all attributes, the map document also contains a list of unlinked attributes in the source and destination business objects.

The following sections describe the format of the two HTML files of a map document.

### **Map-table file format**

The map-table file, *mapDoc.HTM*, describes the map transformations in a tabular format:

- If the map document describes *only one map*, Map Designer creates a single-map map table.
- If the map document describes *two maps*, Map Designer creates a multiple-map map table.

**Single-map map table:** A *single-map map table* describes the mapping flow in a single map; that is, it describes the transformations between a source and destination business object. The single-map map table has the following columns:

- Source Attribute shows the names of the source business object's attributes.
- Transformation Rule describes the kind of mapping transformation between the attribute in the source business object (in the column to the left) and the attribute in the destination business object (in the column to the right). The transformations listed in this column are hypertext links to the location of the attribute in the *mapDocJavaCode.HTM* Java-code file for the map.
- Destination Attribute shows the names of the destination business object's attributes.



Figure 26 shows the HTML file that contains a single-map map table.

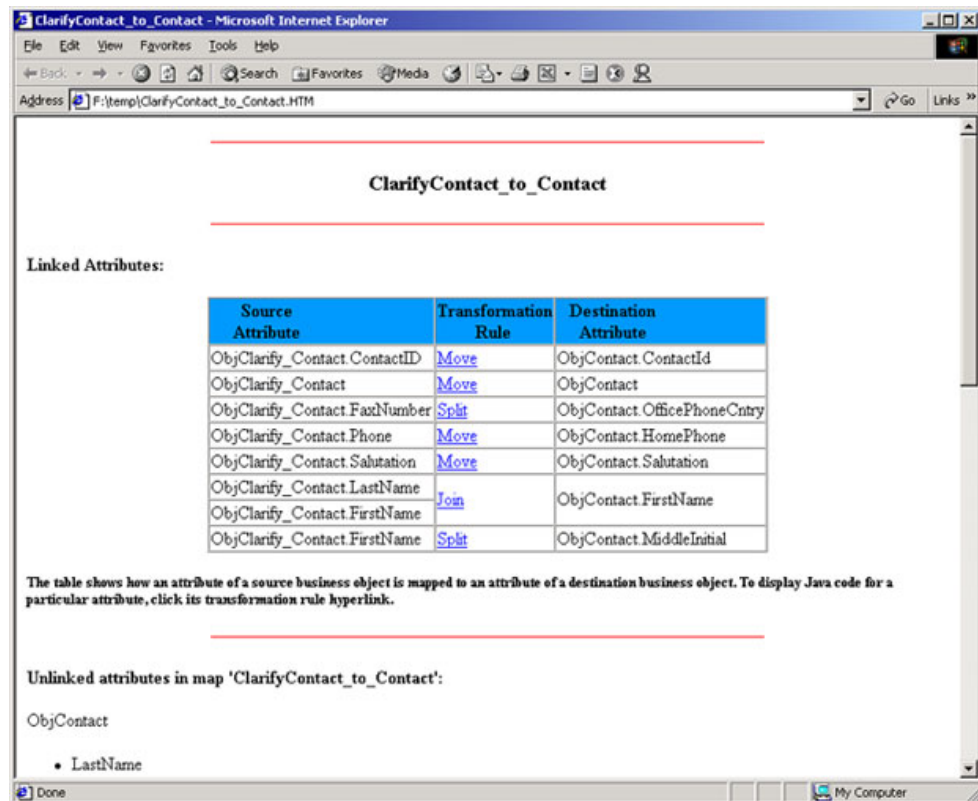


Figure 26. Single-map map table

**Note:** If you enabled the Comment check box Create Map Document dialog, the map table contains a fourth column called Comment, which shows the comment for each of the destination attributes in the table.

**Multiple-map map table:** A *multiple-map map table* describes the mapping flow between two maps; that is, it describes the transformations in the inbound map (between the application-specific and generic business object) and an outbound map (between the generic and application-specific business object). The multiple-map map table has the following columns:

- Source Attribute shows the names of the application-specific business object's attributes.
- The first Transformation Rule column describes the kind of mapping transformation between the attribute in the application-specific business object (in the column to the left) and the attribute in the generic business object (in the column to the right). The transformations listed in this column are hypertext links to the location of the attribute in the *mapDocJavaCode.HTM* Java-code file for the inbound (application-specific to generic) map.
- Common Attribute shows the names of the generic business object's attributes.
- The second Transformation Rule column describes the kind of mapping transformation between the attribute in the generic business object (in the column to the left) and the attribute in the application-specific business object (in the column to the right). The transformations listed in this column are hypertext links to the location of the attribute in the *mapDocJavaCode.HTM* Java-code file for the outbound (generic to application-specific) map.



- Destination Attribute shows the names of the application-specific business object's attributes.

Figure 27 shows the HTML file that contains a multiple-map map table.

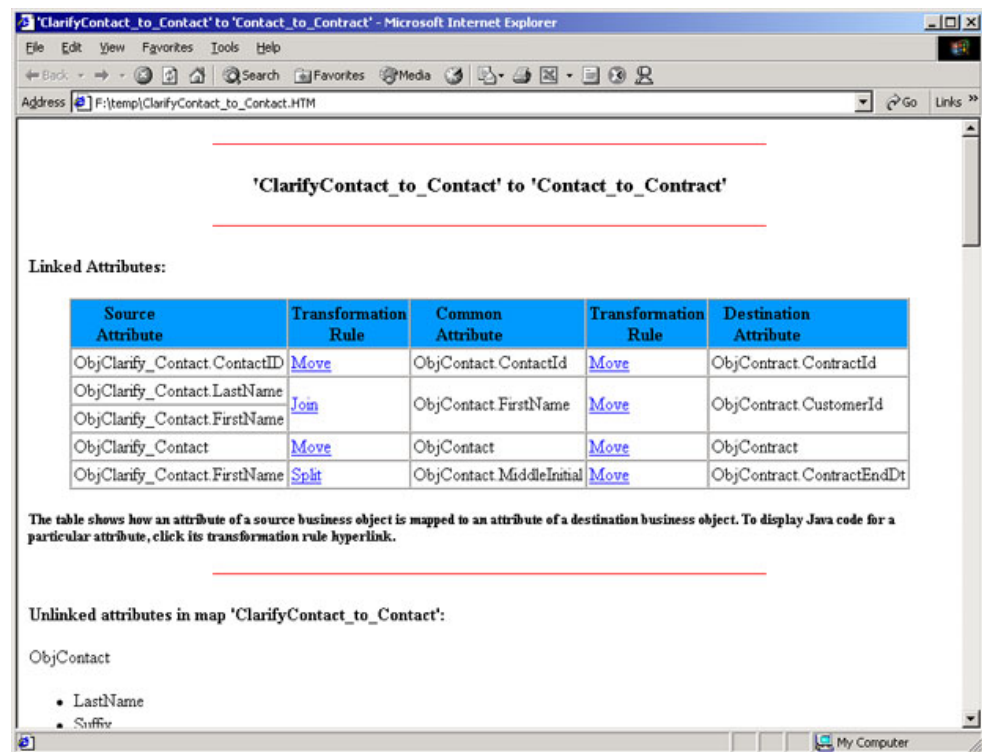


Figure 27. Multiple-map map table

## Java-Code file format

The Java-code file, *mapDocJavaCode.HTM*, provides more detailed information about the map. It contains the Java code that performs the transformations. This code is in standard program format. The Java-code file is useful when you want to view all map transformations in a single operation, rather than opening and viewing each attribute separately.

## Steps for creating a map document

Perform the following steps to create a map document:

1. From the File menu, select Create Map Documents.

**Result:** Map Designer displays the Create Map Document dialog (see Figure 28).

2. Select the map-document configuration options from the Create Map Document dialog:
  - Specify the project.
  - Specify the maps that are involved in the map document.

**Guideline:** If you do *not* select the “Show mapping flow with two maps” check box, you can select only one map from the drop-down list. The drop-down list includes all maps currently defined. If a map is currently open, its name appears by default.

If you select the “Show mapping flow with two maps” check box, the second drop-down list is enabled. This second drop-down list provides only those

maps that share the same generic business object as the first map. From this list, you can select the name of the second map to include in the map document.

- Specify the attributes in the destination business object to include in the map document.

Click the appropriate radio button to indicate whether to include all attributes, only mapped attributes, or only unmapped attributes in the map document.

- Specify a name for the new map document.

**Guideline:** You can click the Browse button to find a location for the map-document file. Map Designer automatically appends the suffix .HTM to the map-document name you enter. Therefore, you do not need to specify a file extension.

3. To initiate creation of the map document, select one of the following options:
  - Click Save to save the selected maps in a map document.
  - Click Save/View to save the selected maps in a map document and view this new map document in an HTML browser.

Figure 28 shows the Create Map Document dialog.

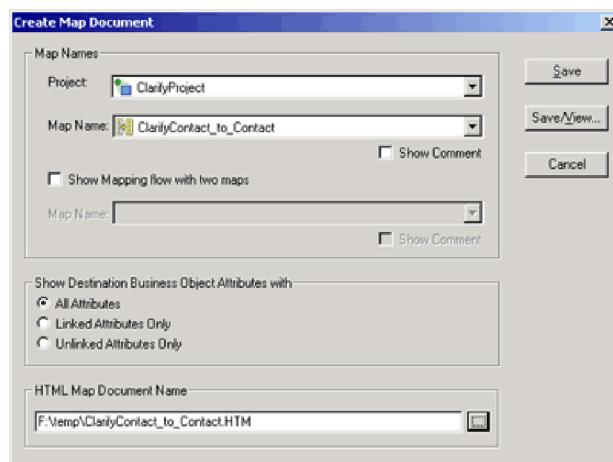


Figure 28. Create Map Document dialog

When you create a map document, Map Designer creates the map document as a Hypertext Markup Language (HTML) file (*mapDoc*.HTM) and a related Java code file (*mapDoc*JavaCode.HTM) where *mapDoc* is the map-document name you specified in the Map Document Configuration dialog.

## Viewing a map document

You can view a map document in any of the following ways:

- Open an existing map document in either of the following ways:
  - From the File menu, select View Map Document.
  - Use the keyboard shortcut of Ctrl+M.

**Result:** The Open dialog displays the available map-document files. Specify the HTML map document to read and click Open.

- Open a new map document by clicking Save/View on the Map Document Configuration dialog.

- Go into the directory that contains the map document files and double-click the desired file.

**Result:** Map Designer invokes your browser to display the HTML map-document file that you selected.

**Guideline:** In addition, you can view the Java code associated with a particular transformation by clicking the entry in the Mapping Action column of the map table. Your browser displays the corresponding Java code segments that implement the mapping between the associated source and destination attributes.

## Printing a map document

Perform the following steps to print a map-document file:

1. View the desired file in your HTML browser.  
For more information, see “Viewing a map document” on page 64.
2. Print the displaying HTML file from the browser by doing one of the following:
  - Select Print from the browser’s File menu.
  - Use the keyboard shortcut of Ctrl+P.
  - Click the Print button in the Standard tool bar.

---

## Using map automation

Map automation allows you to create maps automatically between business objects with similar attributes. You can also generate reverse maps for any given maps.

This section covers the following tasks:

- “Creating maps automatically”
- “Creating reverse maps automatically” on page 68
- “Using synonyms for automation” on page 71

## Creating maps automatically

Map Designer can generate maps automatically between business objects having source and destination attributes with the same names. Even if the business objects are different, they may have certain elements in common. For example, a customer business object usually has the attributes First name, Last name, Address, and Zip code to maintain customer data.

To map business objects automatically, Map Designer looks for attributes with matching names between the source and destination business objects and uses a Move transformation. The mapping happens only at corresponding levels, that is, the top-level attributes in the source business objects are mapped with the top-level attributes in the destination business objects, not any other level. Similarly, the child business objects on the source side are considered for map automation only if corresponding child objects are found in the destination business objects at the same level.

### Steps for creating maps automatically

**Before you begin:** You need to have a map definition file with the source and destination business objects specified. For information on creating a new map definition file with the New Map wizard, see “Steps for creating the map definition” on page 31.

Perform the following steps to create maps automatically:

1. From the Tools menu, select Automatic Mapping.

**Result:** The Automatic Mapping dialog appears, giving you the ability to provide a prefix or suffix for Map Designer to use for searching attributes.

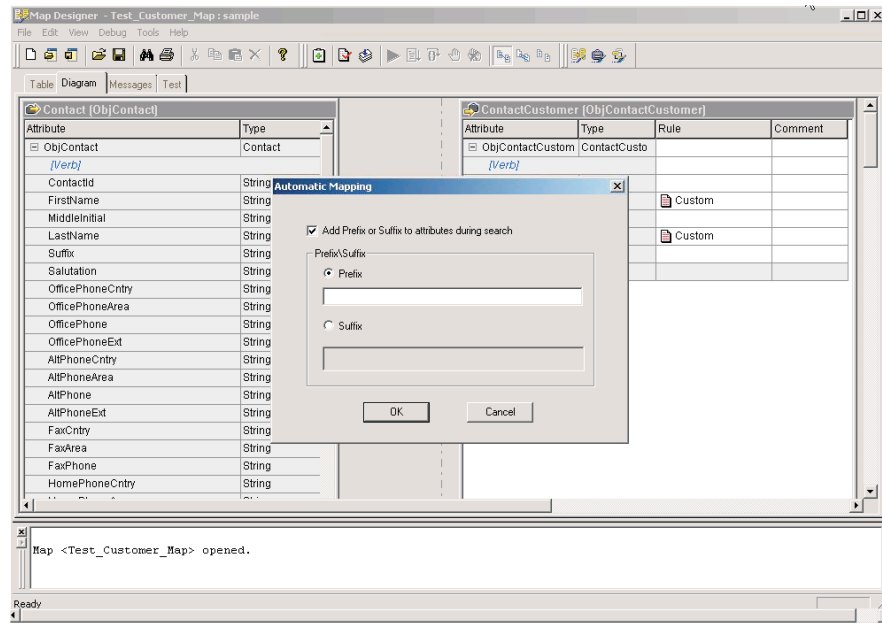


Figure 29. Prefix and Suffix Setting dialog

2. To use this option, do the following in the Automatic Mapping dialog:
  - a. Select the check box Add Prefix or Suffix to attributes during search.

**Note:** This option is disabled, by default.

- b. Select Prefix or Suffix; and in the space provided, type a prefix or suffix to add to the search string for the particular session.

**Restriction:** At any given instance, the choice can only be a suffix or a prefix. You cannot use both at the same time for searching.

- c. Click OK.

**Note:** Map Designer will also use the preferences you have set for case and data types in the Automatic Mapping tab of the Preferences dialog.

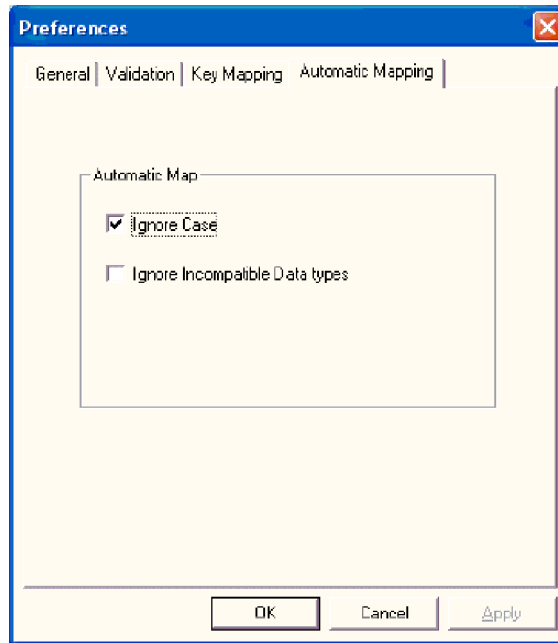


Figure 30. Automatic Mapping tab in Preferences dialog

For information on setting these preferences, see “Specifying Automatic Mapping” on page 23.

**Result:** Map Designer will perform a search on every attribute on the source side with the prefix or suffix added to the search string on the destination side. Every time a matching attribute is found on the destination business object, automatic mapping will take place between the source attribute and the prefixed destination attribute.

### Example of automatic mapping

The following illustration of automatic mapping includes adding a prefix.

Suppose a source business object has the following attributes:

1. FirstName
2. LastName
3. Address
4. Zip

The destination business object has these attributes:

1. ORCL\_FirstName
2. ORCL\_LastName
3. ORCL\_Address
4. Pin
5. State
6. Country

In the Automatic Mapping dialog, we select the check box Add Prefix or Suffix to attributes during search. We type ORCL\_ in the Prefix space and click OK.

**Note:** This example presumes we have previously set the preference to Ignore Case in the Automatic Mapping tab of the Preferences dialog to perform a case-insensitive search on the names.

**Result:** Map Designer performs a case-insensitive search on the attributes on the source side (FirstName, LastName, and Address) with the prefix ORCL added to the search string on the destination side (ORCL\_FirstName, ORCL\_LastName, ORCL\_Address). Every time a matching attribute is found on the destination business object, automatic mapping takes place between the source attribute and the prefixed destination attribute using a Move transformation. In our example, the mapping will occur between FirstName and ORCL\_FirstName, LastName and ORCL\_LastName, Address and ORCL\_Address. The other attributes do not match up, so no mapping takes place between them.

Figure 31 illustrates this example.

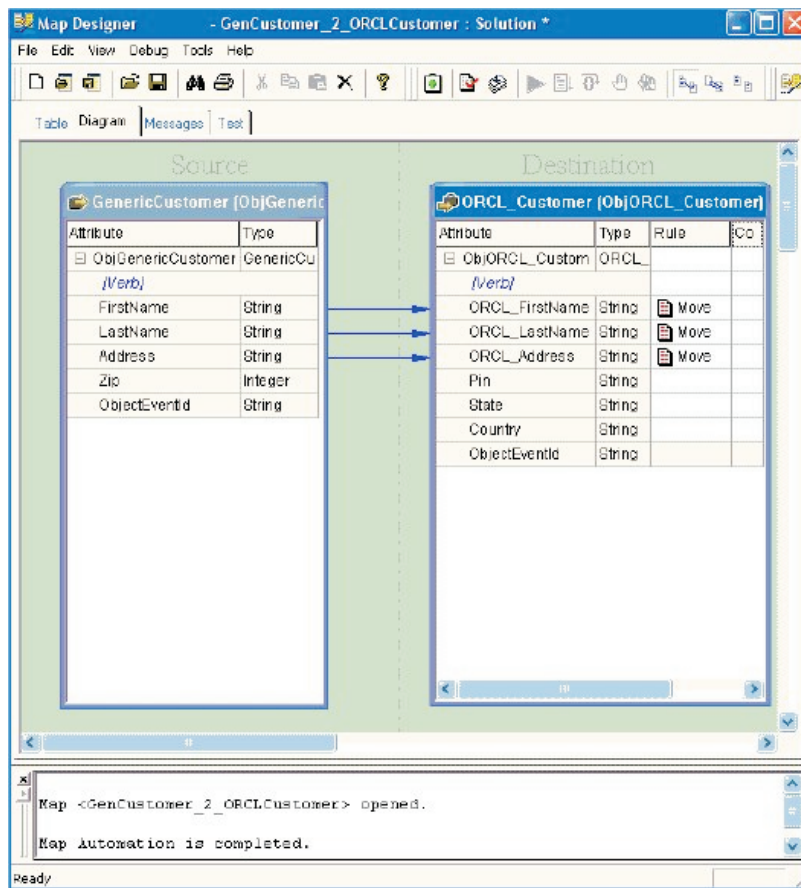


Figure 31. Example of adding a prefix in automatic mapping

## Creating reverse maps automatically

Typically, maps are used in pairs. In most places where a map is used, a map is also needed in the opposite direction. Using Reverse Map, automates the steps required to create a reverse map. The following table shows the standard transformation rules that Map Designer currently supports (Current map column)

and the transformation rules that Reverse Map currently includes (Reverse map column).

Table 20. Transformation rules used for current map to reverse map

Current map	Reverse map
Move	Move
Split	Join
Join	Split
Set Value	No mapping
Custom	No mapping
Cross-Reference	No mapping
Submap	Submap if there is one

As Table 20 shows, reverse mapping presently includes the Move, Split, Join, and Submap transformations. The Set Value, Cross-Reference, and Custom transformation rules are left untouched during a reverse map creation.

**Restriction:** For a Join to Split reverse mapping to take place delimiters must be provided. For a Split to Join reverse mapping, however, delimiters are optional.

### Steps for creating reverse maps automatically

Perform the following steps to create a reverse map automatically.

1. Open the map for which you need a reverse map.
2. From the Tools menu, select Reverse Map.

**Result:** The Save As dialog appears.

3. Type a name for the reverse map and click Save.

**Result:** Map Designer creates a reverse map for the currently open map and opens the reverse map in a new instance of Map Designer.

### Example of reverse mapping

The following example shows a before and after map reversal scenario.

Figure 32 shows a map that needs a reverse map. It uses the Move, Custom, Join, Split, and Set Value transformations.

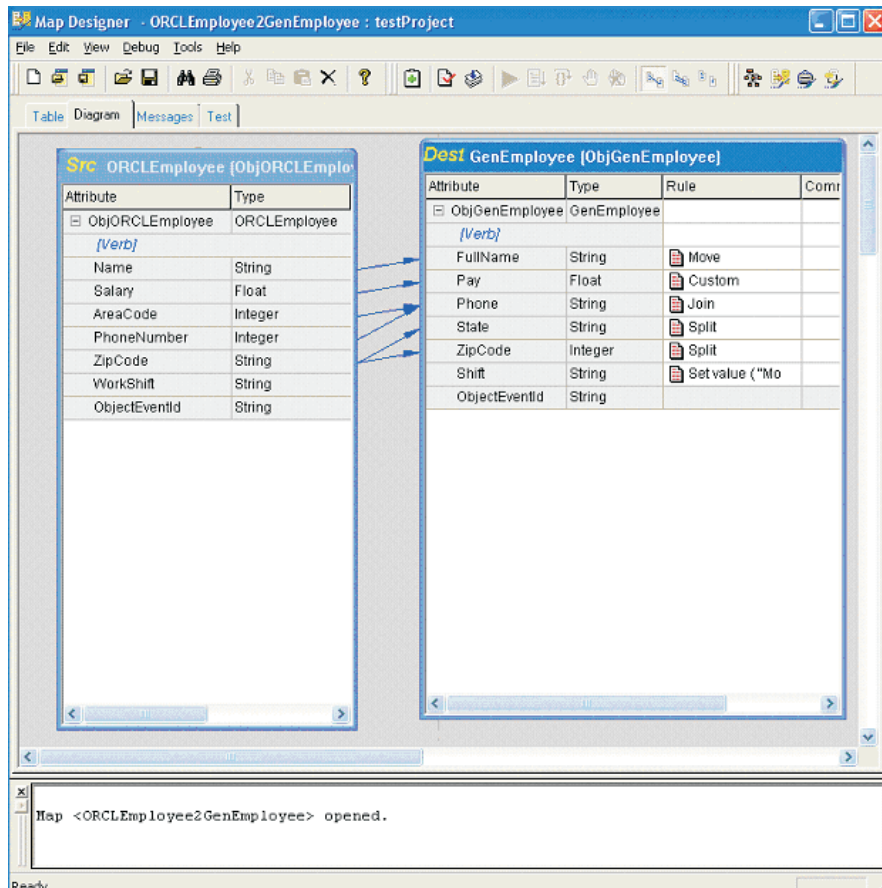


Figure 32. Map that needs a reverse map



After you perform the steps for automatically creating a reverse map (see “Steps for creating reverse maps automatically” on page 69), the following map opens.

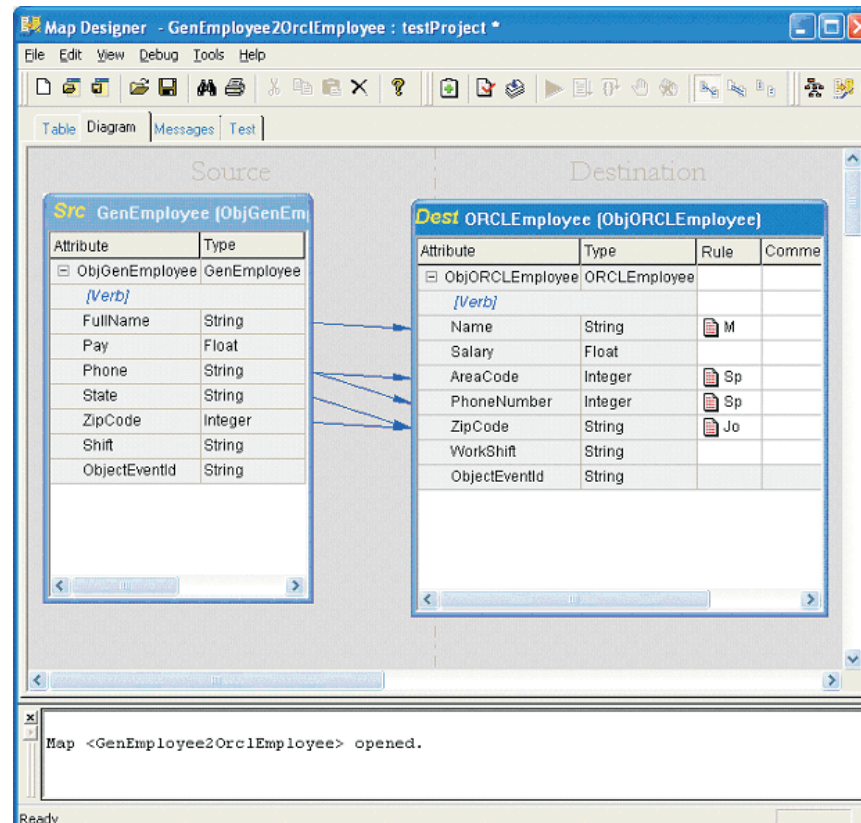


Figure 33. Map created automatically as a result of reversal

As you can see in Figure 33, the Move transformation becomes a Move again in the reverse map. The Split and Join transformations are reversed. The Custom (Pay) and Set Value (Shift) transformations are left untouched. You need to do these manually with Activity Editor. The transformations that cannot be made in the reverse direction will be listed as warnings in the output window.

For information on using Activity Editor, see Chapter 5, “Customizing a map,” on page 103.

## Using synonyms for automation

To enhance the basic matching process, you can create multiple synonyms for an attribute name. For example, you can match an attribute name not only with one matching name but also with several possible equivalent names.

**Example:** Suppose we have a CR as an attribute name on the source side. It could be matched to the following attribute names on the destination side:

- Defect
- Change request
- Bug number
- Defect number
- CR

You add these synonyms at the project level in the Synonyms window of System Manager. You can edit the entries here and add more comma separated strings to help in map automation. You can also create *global* synonyms that apply to all the business objects in the project.

For the procedure for creating synonyms for map automation in System Manager, see the *System Implementation Guide*.

System Manager will search for all the synonyms for a given attribute and perform automatic mapping when it finds the matches. For example, a CR on the source side will match up to *Defect*, *Change request*, *Bug number*, and *CR* if you have added these as synonyms in the Synonyms window. When any of these words is encountered, a mapping will be performed automatically.

---

## Finding information in a map

You can use Map Designer's search facility to perform the following searches:

- Search for text in an attribute name or in the attribute's transformation code.
- Search for unlinked attributes.

### Steps for finding information in a map

Perform the following steps to find information in a map.

1. Initiate a find in one of the following ways:
  - From the Edit menu, select Find.
  - Use the keyboard shortcut of Ctrl+F.
  - In the Standard toolbar, click the Find button.

**Result:** Map Designer displays the Find control pane (see Figure 34).

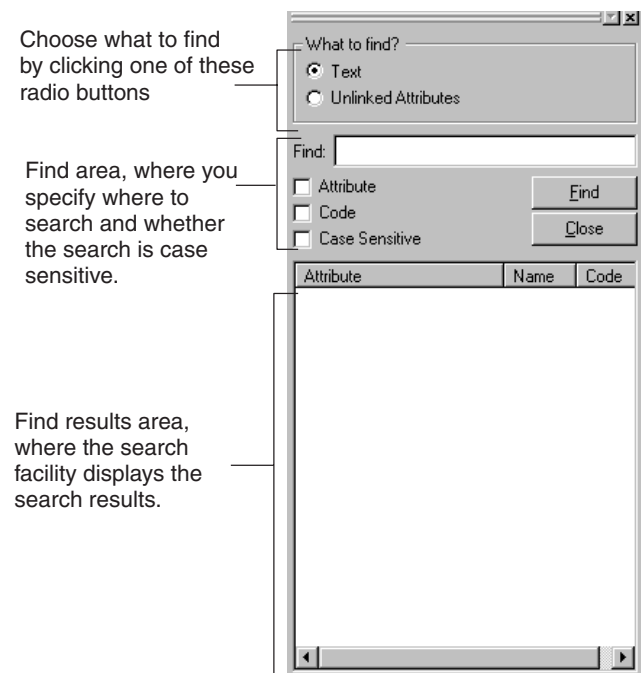


Figure 34. Find Control Pane

2. From the Find control pane, select one of the radio buttons in the What to find? area to indicate which kind of search you want to perform:
  - To search for text:
    - a. Select the Text radio button.
    - b. Enter the text to search for in the Find field. You can enter multiple words and spaces if necessary.
    - c. Indicate where to search for the text by selecting one or more options in the Find area:
 

Attribute—search the attribute names for the specified text.

Code—search the attributes' transformation code for the specified text. You can select either Attribute or Code, or both of those options.

Case Sensitive—make the text search case sensitive. To find only instances of the text that have the same case that you typed, select Case Sensitive.

**Restriction:** You cannot search on data types or comments.
    - d. Click Find to initiate the search.
  - To search for unlinked attributes:
    - a. Select the Unlinked Attributes radio button. The Find control pane deactivates the fields in the Find area.
    - b. Click Find to initiate the search.

**Result:** Map Designer displays the search results in the Find Results area. You can click any attribute name to automatically select that attribute in the map.
3. Click Close to close the Find control pane.

---

## Finding and replacing text

Using Map Designer's Find and Replace capability, you can search for specified text in any customized Java Code or in the comments of a transformation rule (or in both) and replace it with other specified text.

### Steps for finding and replacing text

Perform the following steps to find and replace text.

1. Initiate a find and replace in one of the following ways:
  - From the Edit menu, select Replace.
  - Use the keyboard shortcut of Ctrl+H.

**Result:** Map Designer displays the Replace dialog.

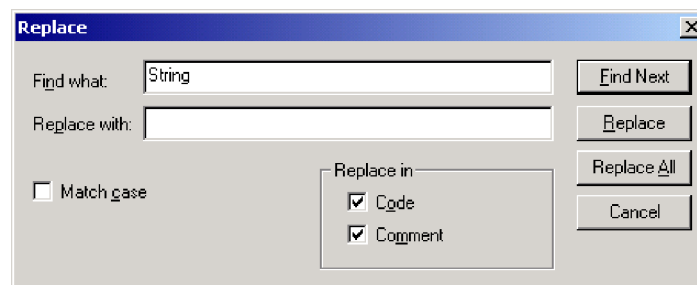


Figure 35. Replace dialog

2. In the Replace dialog, enter the text to search for in the Find what field and the text to replace it in the Replace with field. Select Match case, as necessary.
3. Indicate where to Replace in by selecting either Code or Comment or both.
4. Click Find Next to initiate the search.

**Result:** One of the following results takes place:

- If you specified Replace in Code, when text is found in the customized Java code of a transformation rule, Activity Editor will display with the customized Java code in Quick view mode.
- If you specified Replace in Comment, the Table view will be activated and the text will appear in the comment column in the Table view.

5. Click Replace to replace the match with the new text.

**Guideline:** You can replace all similar matches with one action by clicking Replace All.

6. To continue finding and replacing the specified text, instance by instance, repeat steps 4 and 5.

---

## Printing a map

Map Designer allows you to print a map. It creates a tabular representation of the map, much like the map appears in the Table tab. You can print a map in any of the following ways:

- From the File menu, select Print to print the current map.
- Use the keyboard shortcut of `Ctrl+P`.
- In the Standard toolbar, click the Print button.

Map Designer also supports the following standard print tasks:

- Print Preview—select Print Preview from the File menu to preview the page layout for the current map.
- Print Setup
  - From the File menu select Print Setup to display the Print Setup dialog, where you can configure information such as printer setting, paper size and orientation.
  - Use the keyboard shortcut of `Ctrl+Shift+P`.

**Guideline:** When Map Designer performs the print or print-preview task, it copies the attribute transformation table in the Table tab. Before you print, you can adjust the width of the individual columns and height of individual rows in the attribute transformation table to make the whole map fit on one page or to customize the print result.

---

## Deleting objects

This section provides information on how to delete the following objects:

- “Steps for deleting map transformation steps”
- “Steps for deleting business objects” on page 75
- “Steps for deleting maps” on page 76

### Steps for deleting map transformation steps

Deleting a map transformation step includes three components:

- Deleting the transformation code

- Deleting the comment
- Deleting the data flow arrow

Perform the following steps to delete the transformation step from one of these map tabs.

- From the Table tab: Select the attribute line to delete by clicking in the leftmost column (the column to the left of Exec. Order) and doing one of the following actions:

- Right-click and select Delete Row from the Context menu.
- From the Edit menu, select Delete Current Selection.
- Use the keyboard shortcut of Del.

**Result:** Map Designer automatically deletes any incomplete transformations when you save the map.

- From the Diagram tab: Select the data flow arrow and do one of the following actions:

- From the Edit menu, select Delete Current Selection.
- Use the keyboard shortcut of Del.
- Right-click and select Delete from the map workspace's Context menu.

**Result:** A dialog asks you whether to delete the associated data flow arrow. Click Yes and Map Designer displays a second confirmation asking if you want to delete the associated code. Click Yes and all three items are deleted.

## Steps for deleting business objects

Perform the following steps to delete a business object from a map:

1. Display the Delete Business Object dialog in one of the following ways:
  - From the Edit menu, select Delete Business Object.
  - From the Table tab, perform either of the following actions:
    - Right-click in the empty area of the business objects pane and select Delete Business Object from the Context menu.
    - Right-click the business object in the business objects pane (click the name in the cell) and select Delete <BusObjName> (where *BusObjName* is the name of the selected business object.)

**Result:** The Delete Business Object dialog displays.

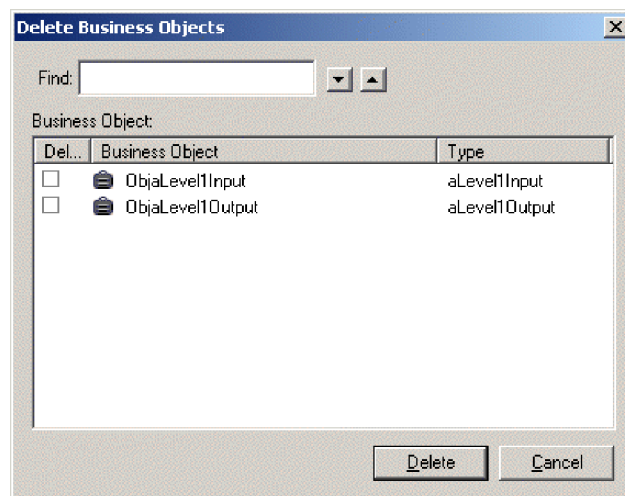


Figure 36. Delete Business Object dialog

2. Through the Delete Business Object dialog, you specify which business objects you want to delete from the map. The Delete Business Object dialog provides the following functionality:
  - To delete a business object:
    - Select the business object in the business object list.
    - Click the Delete button.
  - To locate a particular business object, enter its name in the Find field. The up and down arrows scroll through the business object list.
  - To close the dialog, click Done.

## Steps for deleting maps

Perform the following steps to delete a map from the project in System Manager:

1. From the File menu, select Delete.

**Result:** Map Designer displays the Delete Map dialog, as Figure 37 shows.

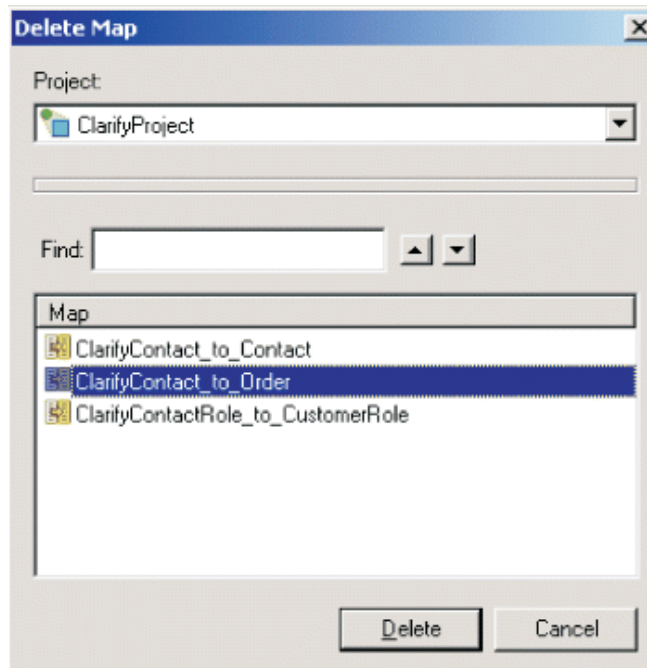


Figure 37. Delete Map dialog

**Note:** If a map is currently open, Map Designer closes this map before it displays the Delete Map dialog. You can specify whether Map Designer closes any currently open map with the option Delete Map: close map before delete. By default, this option is enabled. If the option is disabled, Map Designer provides a confirmation prompt if you select the currently open map from the Delete Map dialog. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

2. Enter the project name.
3. Select the map or maps you want to delete.

From the Delete Map dialog, you can:

- Select a single map by clicking on the map name in the list.
  - Select multiple maps by holding down the Ctrl or Shift key and clicking on the map names.
  - Locate a particular business object by entering its name in the Find field. The up and down arrows scroll through the business object list.
4. Click the Delete button to delete the maps.

**Result:** Map Designer displays a confirmation box for the delete.

**Note:** You can specify whether Map Designer confirms the deletion of a map with the option Delete Map: always display warning message. By default, this option is enabled. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

---

## Using execution order

By default, map execution occurs in the order that the destination attributes appear in the Table tab. Only destination attributes that have transformations are executed. Often, the execution order is the order in which the destination attributes are defined in the destination business object. Figure 38 shows an execution order of the map A-to-B in which destination attributes are executed in the order they are defined.

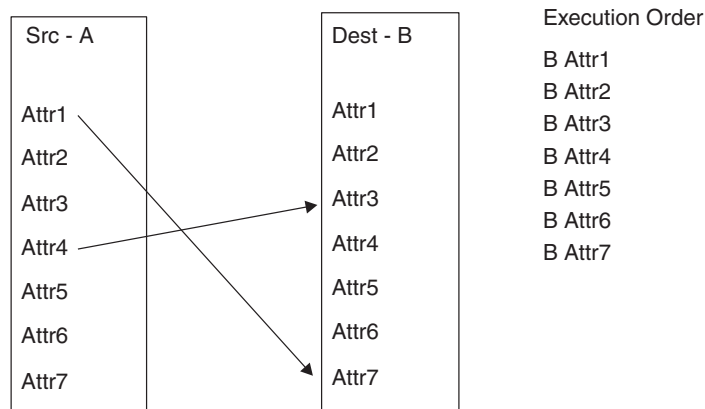


Figure 38. Default execution order

**Note:** Figure 38 assume that all destination attributes have transformation code.

However, certain attributes might have dependencies in their execution order. To ensure that the transformation code of certain attributes is executed before the transformation code of other ones, you can specify the order of their execution. You can change the execution order to specify data flow. For example, suppose in the map A-to-B that Attr7 needs to execute immediately after Attr3 (in other words, Attr7 needs to execute before Attr4). Figure 39 shows how a sequence



specification in the destination business operation changes the sequence.

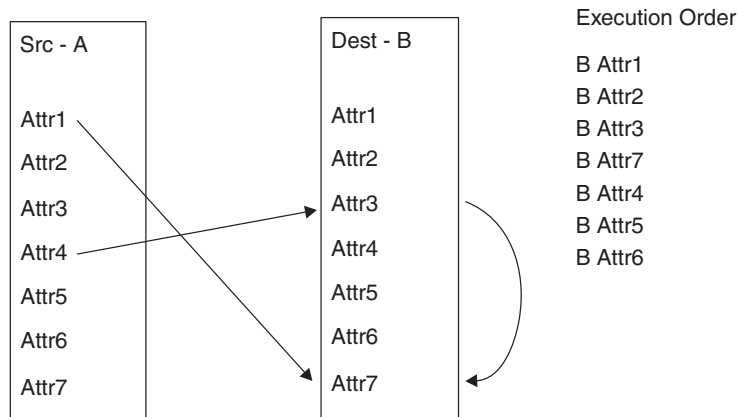


Figure 39. Changing execution order

You can specify an explicit execution sequence that overrides the default order from the Table tab of Map Designer. To specify the sequence of transformations between two destination attributes in the Table tab, click in the Exec. Order field for the destination attribute whose execution order you want to change and enter the desired execution order value.

**Note:** You can specify whether Map Designer rennumbers the execution order for any attributes affected by this change with the option *Defining Map: automatically adjust execution order*. By default, this option is disabled. When the option is enabled, Map Designer automatically adjusts the execution order of other attributes. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

By default, the Table tab displays attributes in the order their transformations are defined. You can then choose to display these mapped attributes by their execution order, their attribute names, or ordered by any other column of the attribute transformation table. Just click the heading of the column to order the attributes by that column’s value.

**Important:** If you click the row header of the transformation and drag-and-drop the transformation to a new position, you change the order in which the transformation rule is displayed. However, this action does *not* affect its execution order.

---

## Creating polymorphic maps

Polymorphic mapping allows a single source business object to map to one of many potential destination business objects. To do this form of mapping, you must:

1. Create a separate map (one source object and one destination object) for each possible outcome.
2. Create a main polymorphic map that has a single source business object and multiple destination objects.
3. Within the first attribute of each destination business object, check some condition that dictates which destination business object is to be populated. If the condition is true, run the appropriate map to accomplish the desired results using the `runMap()` method.



**Example:** Below is sample code from the first attribute in one of the destination business objects in a main polymorphic map. In this example, `ObjInput` is the Instance variable for the source business object, `ObjOutput1` is the Instance variable for the output object which contains this code, and `InputToOutput1` is the submap which performs the actual mapping from `ObjInput` to `ObjOutput1`. In this case, the condition which dictates whether this mapping occurs is based on the value of the `Attr1` attribute within the source business object. Your condition will obviously vary.

```
BusObj[] rSrcBO = new BusObj[1];
BusObj[] rDstBO = new BusObj[1];

rSrcBO[0] = ObjInput;
String Attr1Val = ObjInput.getString("Attr1");

if (Attr1Val.equals("Poly1"))
{
    try
    {
        rDstBO = DtpMapService.runMap("InputToOutput1",
            DtpMapService.CWMAPTYPE, rSrcBO, cwExecCtx);

        ObjOutput1.setContent(rDstBO[0]);
    }

    catch (MapFailureException e)
    {
        e.toString();
        e.printStackTrace();
        raiseException(e);
    }

    catch (MapNotFoundException e)
    {
        raiseException("MapNotFoundException",
            "runMap did not find map");
    }

    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

---

## Importing and exporting maps from InterChange Server

With the `repos_copy` utility, you can load and unload specified map definitions in the repository with the `-e` option. A *map repository file* is the file that the `repos_copy` utility creates when it extracts a map definition from the repository into a `.jar` file. This file contains a map definition in an IBM WebSphere InterChange Server-defined `.jar` format.

**Recommendation:** You should use the `.jar` file extension for the map repository file.

**Example:** The following `repos_copy` command unloads (exports) the `ClCwCustomer` (`ClarifyBusOrg` to generic `Customer`) map definition from the repository of an InterChange Server named `WebSphereICS` into a map repository file:

```
repos_copy -eMap:ClCwCustomer+BusObj:Customer+BusObj:Clarify_Customer
-oNM_ClCwCustomer.jar -sWebSphereICS -pnull -uadmin
```

You can create one repository file that contains all map definition files, including:

- Main map definitions
- Submap definitions
- Files for *both* directions, if applicable.

**Example:** To copy all related map definitions for the ClarifyBusOrg/Customer mapping into a map repository file, use the following `repos_copy` command:

```
repos_copy -eMap:C1CwCustomer+Map:CwC1Customer
-oNM_C1CwCustomer_and_CwC1Customer.jar -sWebSphereICS -pnull -uadmin
```

If you are reusing a submap in several maps, create a separate `repos_copy` file for it instead of putting it in the main text file.

You can also use `repos_copy` to load (import) a map definition into the repository from a map repository file.

**Example:** The following `repos_copy` command loads the `C1CwCustomer` map definition into the repository of an InterChange Server named `WebSphereICS`:

```
repos_copy -iNM_C1CwCustomer.jar -sWebSphereICS -uadmin -pnull
```

This `repos_copy` command assumes that the `C1CwCustomer` and `CwC1Customer` map definitions do *not* currently exist in the repository. If they do exist, this command fails to load these new map definitions. You can use one of the `-a` options of `repos_copy` to choose how to handle duplicate map definitions:

---

<code>-ai</code>	Skip over duplicate map definitions during the load
<code>-ar</code>	Overwrite any duplicate map definitions with the map definition in the map repository file.
<code>-arp</code>	Interactively query the user whether to overwrite any duplicate map definitions with the map definition in the map repository file.

---

**Note:** In Production mode, the maps will be automatically compiled.

You can also use `repos_copy` to load and unload relationship definitions in the repository. For more information, see “Loading and unloading relationships” on page 309.

---

## Converting old maps

Map design content is no longer stored or saved in `.dlm` files in the file system. Instead, it is stored in the repository in XML format, and is no longer accessible from the file system. To view, modify, or compile a map from a release prior to 4.2.0, you must convert the map to the new XML format.

**Note:** You need to upgrade any map prior to 4.2 before you can update its map properties dynamically with System Manager.

Map Designer provides a conversion utility for 4.0 and 4.1.x maps that facilitates conversion of individual or sets of maps. When you attempt to open a map created from a previous version, Map Designer prompts you to convert the map to the new format.

Click Yes to automatically convert the old map to the new format so it is readable in Map Designer.

**Recommendation:** When the conversion completes, it is good practice to visually inspect the new map for correctness.

To save the converted map, you must explicitly save it with one of the options of the Save submenu in the File menu. Otherwise, the map remains in the old format and is not readable by Map Designer.

Map Designer only supports converting 4.0 and 4.1.x maps that are in .cwm format. For .dlm map format, you must use the repos\_copy utility to import the map definition into InterChange Server. If you want to modify or view the content of such maps, you must convert them by importing these maps into a project with System Manager.

**Note:** Existing map instances (compiled map definitions) do not require conversion. Only if the behavior of a map must be modified or the map must be regenerated must you convert its map definition to the new format and recompile it.

For more information, see the *System Implementation Guide*.



---

## Chapter 4. Compiling and testing maps

This chapter describes how to validate, compile, and test maps using Map Designer.

The chapter covers the following tasks:

- “Checking the transformation code” on page 83
- “Validating a map” on page 84
- “Compiling a map” on page 84
- “Compiling a set of maps” on page 86
- “Testing maps” on page 87
- “Doing advanced debugging” on page 94
- “Testing maps that contain relationships” on page 95
- “Debugging maps” on page 100

---

### Checking the transformation code

When you have finished writing the transformation code associated with a destination attribute, you can perform a limited syntax check on the code. By checking as you proceed, you reduce debugging time required at the end of the map development process. You can check attribute code using the technique of finding unmatched delimiters.

**Note:** This technique is also useful when you have a compilation error whose cause you cannot immediately determine from the error message.

### Finding unmatched delimiters

Map Designer provides the Check for Unmatched Delimiters feature to help you resolve one of the errors in a program that is most difficult to identify. This feature checks for unmatched delimiters in an attribute’s transformation code. Map Designer checks for these paired tokens: ( ), [ ], { }, “ ”, and ‘ ’.

#### Steps for finding unmatched delimiters

Perform the following steps to do a syntax check on an attribute’s transformation code:

1. Invoke Activity Editor in Java mode.  
For information on how to display Activity Editor, see “Starting Activity Editor” on page 103.
2. Use the Check for Unmatched Delimiters option in Activity Editor. Right-click and select Check for Unmatched Delimiters from the Context menu.

**Note:** If an unpaired instance of one of the delimiters exists, Activity Editor displays a message in the output window, providing the line number where the error was unable to be resolved. This line number might not be the actual line of the missing delimiter.

3. To go to the source of the unmatched delimiter, note the line number displayed at the bottom of the window.

**Tip:** To move to this line, use the Goto Line option from either the Edit menu or the Context menu of Activity Editor. Enter a line number to navigate to the line where the problem occurred.

**Note:** If the problem is caused by unmatched quotation marks at one end of a string literal, the string does not appear pink as it should. When you add the missing quotation mark, the entire string turns pink.

---

## Validating a map

Map Designer's validation process verifies the accuracy of the map's data flow by performing the following checks:

- Ensures that the map has no incomplete transformation steps.
- Ensures that indexes to business object arrays are properly sequenced, starting from zero (0).
- Provides a warning if any transformation step maps to the `ObjectEventId` attribute.
- Validates transformations:
  - Makes sure execution order is correct; that is, that execution order is unique, positive, and consecutive.
  - Ensures that no attributes have cyclic dependencies on each other. If any cyclic transformations are found, Map Designer displays the cyclic rules in the output window.
  - Checks transformation information:
    - Move transformation—only one source attribute is involved.
    - Join transformation—more than one source attribute is involved.
    - Split transformation—only one source attribute is involved; split index is greater than or equal to zero; split delimiter is not empty.
    - Set Value transformation—no source attribute is involved; a value has been specified.
    - Submap transformation—at least one source attribute is involved; submap name is specified.
    - Cross-Reference transformation—only one source attribute is involved.

Map Designer automatically validates a map when you save it. You can also choose to validate the map by performing either of the following actions:

- From the File menu, select Validate Map.
- In the Designer toolbar, click the Validate button.

At this point, if you have specified any options on the Validation tab of the Preferences dialog, Map Designer will issue a warning if the specific condition is not mapped.

For more information on setting dependencies between attributes, see "Using execution order" on page 77.

---

## Compiling a map

When it compiles a map, Map Designer generates a `.class` file from the `.java` file that holds Java code for the map's transformations. It generates this `.java` file from the transformation code stored as part of the map definition in the project.

**Important:** To be able to compile a map, the Java compiler (javac) must exist on your system and its path must be on your PATH system variable. For more information, see “Setting up the development environment” on page 11..

## Steps for compiling a map from Map Designer

From within Map Designer, you can initiate compilation of a map in several ways:

- To compile the *current* map, do one of the following:
  - From the File menu, select Compile.
  - Use the keyboard shortcut of F7.
  - In the Designer toolbar, click the Compile button.
- To compile the *current map and any submaps* that this map is using:
  - From the File menu, Select Compile with Submap(s).
- To compile *all* or a subset of maps defined in System Manager, do one of the following:
  - From the File menu, select Compile All.
  - Use the keyboard shortcut of Ctrl+F7.

For more information, see “Compiling a set of maps” on page 86.

By default, Map Designer saves the map in the project before it begins the compile and generates the Java code in the .java file and .class file. If any message file is needed, Map Designer will also generate the message file.

**Note:** You can specify whether Map Designer automatically saves a map to the project before compiling the map with the option `Compile Map: save map before compile`. By default, this option is enabled. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

To compile, Map Designer calls the Java compiler on the map’s Java source code (.java file). The actions it then takes depend upon whether the compilation is successful.

## Steps for compiling a map from System Manager

System Manager also provides several ways to compile a map:

- To compile a single map, do one of the following:
  - Highlight the desired map and select Compile from the Component menu.
  - Right-click the desired map and select Compile from the Context menu.
- To compile a map and its submaps:
  - Right-click the desired map and select Compile with Submap(s) from the Context menu.
- To compile *all* maps defined in the project:
  - Highlight the Maps folder and select Compile All from the Component menu.

**Note:** You will need to select which map folder in the project to compile all maps for by right-clicking on the map folder and selecting Compile All from the Context menu.

## A successful map compilation

When the map successfully compiles, Map Designer takes the following steps:

- Compiles the Java code into a .java file.
- Displays the following message in the output window at the bottom of each Map tab to indicate that there are no errors during compilation:  
Compilation is successful.

## An unsuccessful map compilation

If an error occurs during compilation, Map Designer generates error messages and displays them in the output window at the bottom of the screen. Unless an output window is already open, Map Designer opens one at the bottom of the Map tab to display these compilation messages.

When a compile error occurs, the output window displays the error message with the problematic attribute name and line number in blue. Click the hyperlink to navigate to the problematic area in the Java view in Activity Editor.

**Tip:** You can clear the output window of messages by selecting Clear Output from the View menu.

Some errors are easy to detect, while others are not.

---

## Compiling a set of maps

Using the Compile All option on the File menu, you can compile all maps in your System Manager, or a subset of maps.

### Steps for compiling a set of maps

Perform the following steps to compile a set of maps:

1. From the File menu, select Compile All.

**Result:** Map Designer displays the Compile All Maps window.

2. Select the project for the map compile.
3. Select the maps to compile.

**Guideline:** Selecting any check box at the root will automatically check all its child check boxes. Thus, when you select a project, all maps in that project are selected. To select only a subset of maps, deselect the appropriate Compile check boxes.



Figure 40 illustrates the Compile All Maps window.

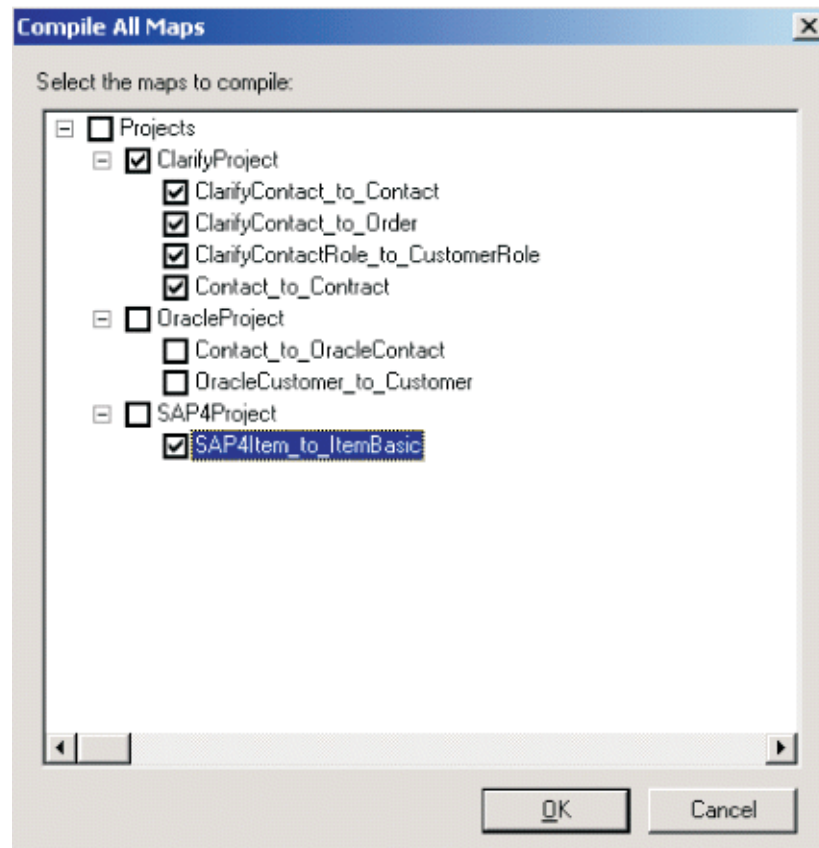


Figure 40. Compile All Maps window

**Result:** Map Designer displays the success or failure of each map's compilation in the output window. You might want to enlarge the size of the output window before starting the compilation process so you can see more of the compilation status messages.

---

## Testing maps

You can test a map's transformation steps by providing sample data for the source business object and executing a test run of the map. A *test run* is map execution that does not involve an event sent by a connector or a call sent by an access client; the map executes within Map Designer. Map Designer provides a separate tab, the Test tab in the Map Designer window to test maps and view test results.

**Note:** When a map is selected from Testing Environment for further debugging, Testing Environment will launch Map Designer, giving Map Designer the input business objects to the map under testing.

This section describes how to set up and execute a test run, using these main steps:

- "Steps for preparing to run the test" on page 88
- "Creating test data" on page 88
- "Setting breakpoints" on page 90
- "Running the test map" on page 92
- "Viewing test run results" on page 93

- “Steps for changing the map and re-executing” on page 94

**Tip:** An alternative testing strategy, which is not covered in detail, is to set breakpoints in the map and to send a triggering event from the connector, which causes the map to execute.

## Steps for preparing to run the test

Before running the test, perform the following steps:

1. Open the map to debug from the project.
2. If the map has *not* been compiled since the last modification, compile it by selecting Compile from the File menu. For more information, see “Compiling a map” on page 84.
3. If the Test tab of Map Designer is *not* currently displaying in the tab window, select the Test tab.

## Creating test data

Every time you test a map, you must load data into the source business object. To do this, use the Source Testing Data pane in the Test tab (see Figure 41). The Source Testing Data pane allows you to specify the following test information:

- The calling context—indicates the map execution context for the map run.
- The generic business object—provides test data for the generic business object when testing the SERVICE\_CALL\_RESPONSE calling context for an identity relationship.
- The test data—data for the attributes of the source business object.

**Important:** The calling context and generic business object are required *only* for testing relationships within maps. For more information, see “Testing maps that contain relationships” on page 95.

## Testing the map for the first time

When you test the map for the first time, you must manually enter the values of the attributes in the Source Testing Data pane.

The following sections provide information about how to enter this data:

- “Guidelines for creating test data for the source business object” on page 88
- “Steps for creating test data for a child business object” on page 89

**Guidelines for creating test data for the source business object:** To create source business object data for the first time, follow these rules:

- To set the verb, select it from the verb combo box in the verb row.
- To assign a value to a source attribute, type it into the attribute’s Value column. You do *not* have to provide values for all attributes.
- To assign a value to a relationship attribute, specify the appropriate value in the Value column and make sure you also specify the correct calling context. For more information, see “Testing maps that contain relationships” on page 95.
- To assign values to a child business object, right-click the child object and select Add Instance from the Context menu. For more information, see “Steps for creating test data for a child business object” on page 89.
- To assign default values to the source attributes attribute, select the source business object and select Reset from the Context menu.
- If you are testing relationships, make sure to set the ObjectEventIds of the source parent object and all child objects that participate in the relationships.

- To save the values you have entered for future test runs, create a business object (.bo) file by selecting the source business object and performing either of the following actions:
  - Click the Save To button in the Source Testing Data pane.
  - Select Save To from the Context menu. When prompted, enter a file name where these values will be stored.

**Result:** The next time you test this map, you can click the Load From button and the attributes will be filled in automatically from the business object file.

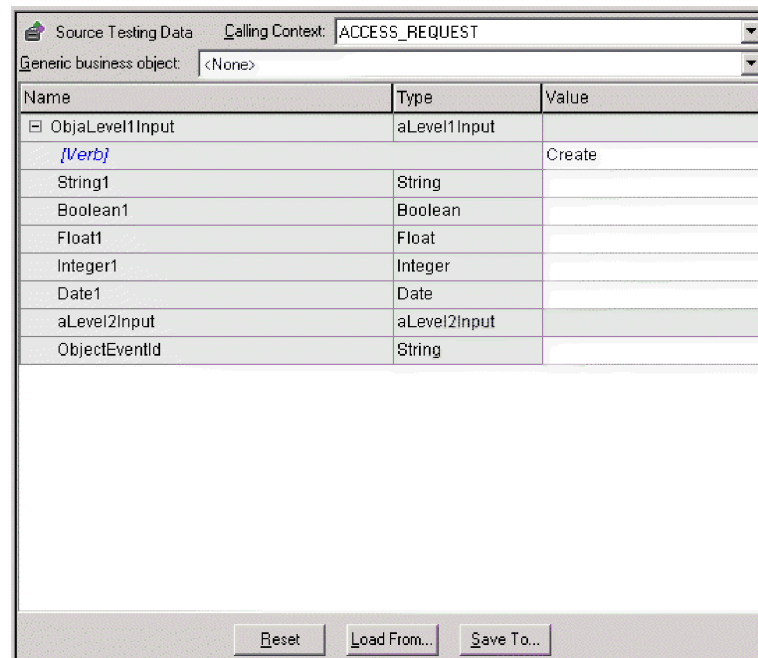


Figure 41. Source Testing Data pane of the Test tab

**Steps for creating test data for a child business object:** If the source business object has child business objects and you want to specify test data for the child attributes, you must first create an instance for each child object you need. To do this, perform the following steps:

1. Right-click the child business object name and select Add Instance from the Context menu. When you expand the object, you see the instance that Map Designer has created.
 

**Guideline:** The first instance you add has an index number of zero. You can have as many instances as you want (as long as the child attribute has multiple-cardinality).
2. Click the plus symbol (+) beside the instance index number to expand the child business object.
 

**Result:** When you expand the object, you see the child attributes for this instance.
3. To create data for the child business object instance, follow these guidelines:
  - To set the verb for the child business object, select it from the verb combo box in the verb row.
  - To specify a value for a child attribute, select it and enter the value in the Value column.

- If the name of the attribute is followed by (N), the attribute contains a multiple-cardinality child business object and you can add more instances. To add a child business object to the end of the array, right-click the last index and select Add Instance from the Context menu.
- Modify the values of as many instances as you want. Add and remove instances as follows:
  - To add an instance, right-click the child instance name and select Add Instance.
  - To delete an instance, right-click the instance name of the child instance you want to delete and select Remove Instance.
  - To delete *all* instances, right-click the child instance name and select Remove All Instances. This option is only enabled if the child business object has multiple-cardinality.

### Testing the map in subsequent runs

For subsequent test runs, Map Designer reuses the previously specified test data. You can perform any of the following actions on this data:

- Leave all test data as it is.
- Modify values for any individual attributes by changing the appropriate entries of the Values column.

**Tip:** If you modify the data, remember to resave any business object (.bo) file.

- Load a set of values from a business object (.bo) file.

To load attribute values from a business object file, select the source business object and perform either of the following actions:

- Click the Load From button in the Source Testing Data pane.
- Select Load From from the Context menu.

When prompted, enter the name of the business object file to be loaded.

- Return all source destination values to their defined default values by selecting the source business object and selecting the Reset option from the Context menu.

## Setting breakpoints

When you set a breakpoint, map execution pauses just before the transformation of the destination attribute on which the breakpoint is set. The use of breakpoints lets you step through map execution and check the sequence and the results of individual operations. You can set as many breakpoints as you like.

**Guideline:** Breakpoints are not part of the map's definition. You set breakpoints on the map after the map is opened in Map Designer, and when the map is debugged (either with Debug > Run Test or Debug > Advanced > Attach). Breakpoints have no effect on the map when the map is not debugged from Map Designer.

**Note:** You can only set a breakpoint on a destination attribute that has a transformation defined for it.

### Steps for setting breakpoints

Perform the following steps to set a breakpoint.

1. Use one of the following methods:
  - Right-click a destination attribute in the Destination Testing Data pane and select Set Breakpoint from the Context menu. If the destination source attribute is not yet expanded, you can expand it with either of the following commands:

- Click the plus symbol (+) next to the destination business object.
- Select the destination business object and select Expand from the Context menu.

**Note:** The Context menu of the destination business object also provides a Collapse option.

- Select Toggle Breakpoint from the Debug menu.
- Use the keyboard shortcut of F9.
- In the Designer toolbar, click the Toggle Breakpoint button.

**Note:** The Toggle Breakpoint option toggles a breakpoint definition on and off. If the breakpoint is *not* currently set, Toggle Breakpoint sets it. If the breakpoint is currently set, Toggle Breakpoint removes it.

**Result:** Map Designer displays a dark circle next to the destination attribute on which the breakpoint is set, as shown in Figure 42.

Name	Type	Value	Rule	Source Attribute	Com
[-] ObjLevel1Output	aLevel1	{Local}	[Cross Ref]	ObjLevel1Input	
[Verb]			[Move]	ObjLevel1Input [Verb]	
● String1	String		[Move]	...String1	
Boolean1	Boolean		[Join]	...String1, ...Boolean1	
Float1	Float		[Split]	...Float1	
Integer1	Integer		[Set Value]		
Date1	Date		[Custom]		
[+] aLevel2Output	aLevel2	{Local}	[Submap]	...aLevel2Input	
ObjectEventId	String				

Figure 42. Breakpoint set

Once you set the breakpoint, the execution of the map instance pauses at this breakpoint and you can see the current status of the map. Unless you specify at least one breakpoint, the map executes and finishes with the message:

Test run finished

**Rule:** You must always provide values for the source data associated with the destination attributes where you set the breakpoints. Otherwise, the transformation rule will run normally and the breakpoints will execute normally, but the destination value will usually be empty, depending on what transformation rule is defined. For more information, see “Creating test data” on page 88.

To view all breakpoints for the map, select Breakpoints from the Debug menu.

**Result:** Map Designer displays the Breakpoints dialog (see Figure 43).

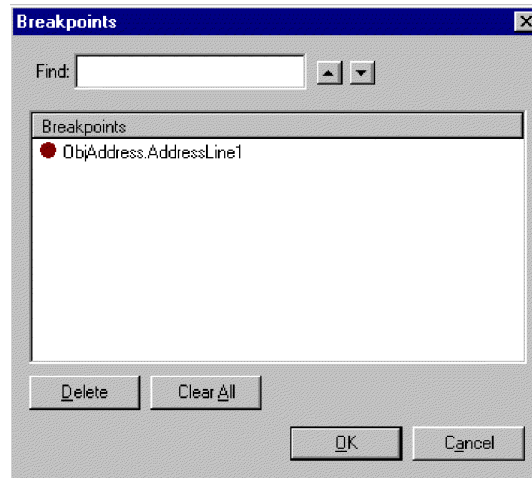


Figure 43. Breakpoints dialog of the test tab

2. From the Breakpoints dialog, you can perform any of the following actions:
  - Locate a destination attribute on which a breakpoint is set—double-click the breakpoint name.

**Tip:** To locate a particular breakpoint, enter its name in the Find field. The up and down arrows scroll through the business object list. In the Destination Testing Data pane, Map Designer highlights the destination attribute.

- Remove a breakpoint—in the Breakpoints area, select the breakpoint to remove and click the Delete button.

You can also remove a breakpoint by performing any of the following actions:

- Right-click a destination attribute in the Destination Testing Data pane and select Clear Breakpoint from the Context menu.
  - Use any of the commands for the Toggle Breakpoint option on an existing breakpoint. For more information, see “Setting breakpoints” on page 90.
- Clear all breakpoints that display in the Breakpoints area—click the Clear All button.

You can also clear all breakpoints by performing any of the following actions:

- From the Debug menu, select Clear All Breakpoints.
- In the Designer toolbar, click the Clear All Breakpoints button.

## Running the test map

Once you have entered the source test data and set any desired breakpoints, you are ready to test the map. To run a map test involves:

1. “Steps for starting the test run”
2. “Steps for processing breakpoints” on page 93 (if any breakpoints have been set)

### Steps for starting the test run

To start the test run, perform the following steps:

1. Perform one of the following actions:
  - From the Debug menu, select Run Test.

- In the Designer toolbar, click the Run Test button.

**Result:** The Connect to IBM WebSphere InterChange Server dialog box will display and allow you to connect to the server for testing.

2. In the dialog, enter the server name, user name, and password.
3. Specify whether you want to deploy the map and dependent business objects for the test run.

**Guideline:** Deploying a minimum set of business objects to the server for testing will minimize debugging initialization time.

**Result:** Execution of the map starts. Map Designer displays the following message in the output window:

Starting test run...

### Steps for processing breakpoints

Map execution pauses when it reaches a destination attribute where you have set a breakpoint. When the breakpoint is reached, Map Designer takes the following actions:

1. Highlights the destination attribute on which the breakpoint was set and displays a dark circle with a yellow arrow next to it.
2. Displays the following message in the output window:

Test Run stopped at attribute *AttrName* (next transformation > "Rule").

**Tip:** With map execution paused, you can examine the values of the destination attributes that have been processed so far by looking in the Value column of the Destination Testing Data pane.

3. Processes the breakpoint and continues map execution, when you do either of the following actions:
  - Proceed to the next breakpoint or the end of the map, whichever comes first.
 

To continue map execution, perform one of the following actions:

    - From the Debug menu, select Continue.
    - Use the keyboard shortcut of F8.
    - In the Designer toolbar, click the Continue button.
  - Execute this destination attribute, then stop before executing the next attribute.
 

To continue map execution for only one more step, perform one of the following actions:

    - From the Debug menu, select Step Over.

**Tip:** Select this option to watch the code execute attribute by attribute.

    - Use the keyboard shortcut of F10.
    - In the Designer toolbar, click the Step Over button.

**Result:** When the execution of the test run is finished without any run-time errors, Map Designer displays the following message in the output window:  
Test run finished.

## Viewing test run results

Test run results display in the destination business object, which is in the Destination Testing Data pane. Values resulting from the map transformations are visible in the Values column of this table. You can view test run results by either:

- “Watching the process” on page 94
- “Viewing results after execution” on page 94



## Watching the process

During a test run that has test data and breakpoints, you can watch as the destination business object fills with values. Values appear in the Values column in the Destination Testing Data pane as they are processed. When map execution is paused on a breakpoint, all destination attributes *before* that attribute in the execution order have values displayed.

To view the transformations as they occur:

- Set a breakpoint on the second destination attribute and step through map execution with the Step Over option. The map will be read-only.

## Viewing results after execution

To view test run results when the map has already executed, examine the destination business object in the Destination Testing Data pane.

To save the test results:

- Highlight the destination business object and select Save To from the Context menu.

**Result:** Map Designer saves the values of the destination attributes in a business object (.bo) file.

## Steps for changing the map and re-executing

As you test the map, you might discover the need to change the map. To edit the map and then continue the test, perform the following steps:

1. Switch to either the Table or Diagram tab to view the map transformations.
2. Make the edits to fix the errors.
3. Recompile the map.
4. Continue the testing process by switching back to the Test tab.
5. Begin a new test run.

### Important:

1. Make sure you complete the test run, either with success or failure, before you attempt to recompile the map.
2. After you modify the map, be sure to deploy the map to the server for the change to be reflected in the server.

---

## Doing advanced debugging

Besides debugging maps that are stored in local projects, you can also directly debug a map that resides in the server. Perform the following steps to do this:

1. Select Debug > Advanced > Attach.

**Result:** The Connect to WebSphere InterChange Server dialog displays.

2. Enter the Server name, User name, and Password; and click Connect.

**Result:** Map Designer displays a list of new maps on that server.

3. Select the map you want to attach to.

**Result:** The map opens in Map Designer in Read-only mode.

4. Set breakpoints in the map to have the server pause map execution at a certain transformation rule.

**Result:** When a breakpoint is hit on the server, you can step over or continue map execution, as usual. The resulting business object values will display in the Destination Test Data pane.



5. Stop the debugging session at any time using Debug > Advanced > Detach.  
**Result:** Map Designer will close the map.

---

## Testing maps that contain relationships

When you test a map that contains a relationship transformation, you need to provide the following information in addition to the test data:

- The calling context

Part of a map's execution context includes a calling context. Many of the relationship methods in the Mapping API use this calling context to determine what action to take during the mapping. For this reason, if you are testing a relationship attribute in a map, you usually must specify the appropriate calling context for the transformation.

- The generic business object definition

When you test the `SERVICE_CALL_RESPONSE` calling context for an identity relationship, you need to specify the map's generic business object so that the test run can locate the generic key value in the relationship.

**Note:** For more information on calling maps within a collaboration, see "*Calling a native map*" section in the *Collaboration Development Guide*

You specify this information in the Source Testing Data pane of the Test tab.

**Tip:** If the width of the Source Testing Data pane is not enough to let you see the complete menu options of the Calling Context combo box, you can expand the size of this area by putting the cursor over the right-hand boundary until you see the following symbol `<-||->` and drag the boundary to the right.

If you are testing Relationships, select the appropriate generic object from the list of business objects, select Calling Context, and set the `ObjectEventIds` for the parent and child objects that match the ones you already set in the Test Data screen. The calling context you need to provide and whether you need to specify a generic business object depend on the type of relationship you are testing. This section provides information on the following:

- "Testing an identity relationship"
- "Testing a lookup relationship" on page 98

## Testing an identity relationship

To test point-to-point mapping (from Application 1 to Application 2) for an identity relationship you use three maps:

- An inbound map from Application 1's application-specific business object to a generic business object—`App1_to_Generic`
- An outbound map from the generic business object to Application 2's application-specific business object—`Generic_to_App2`
- An inbound map from Application 2's application-specific business object to the generic business object—`App2_to_Generic`

**Example:** Figure 44 shows an example of a point-to-point communication of customer data between a Clarify application and an SAP application. If each application uses a unique key value to identify customers, these three business objects can be related with an identity relationship. Therefore, each map includes a

cross-reference transformation rule. As each of these maps executes, these relationship methods access the calling context to determine the actions to take.

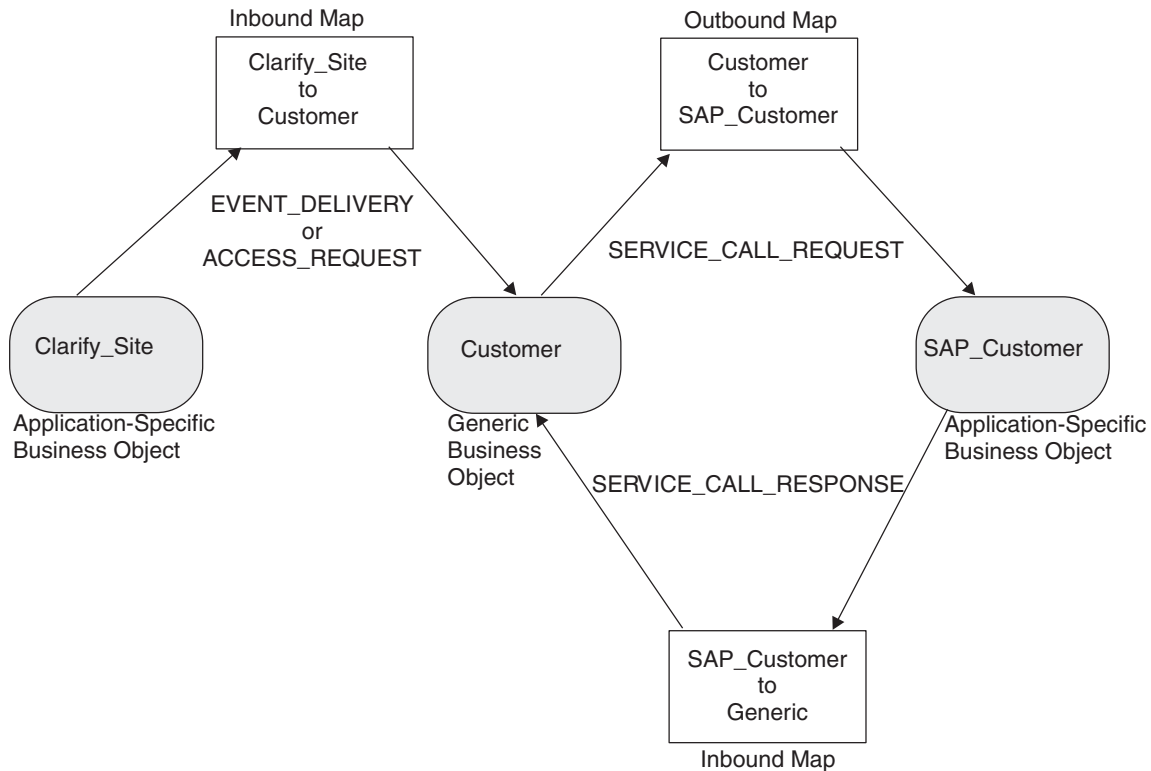


Figure 44. Maps involved in point-to-point testing of an identity relationship

To test the Create verb, you need to verify that a new application-specific key value in Application 1 (Clarify application in Figure 44) causes a new generic key value to be added for the generic business object *and* a new application-specific key value in Application 2 (SAP application in Figure 44). Therefore, testing involves three steps:

1. Testing the inbound map, App1\_to\_Generic, to send in a new key value from Application 1 and ensuring that a new key value is generated for the generic business object. Follow the steps in Table 21.

Table 21. Testing the App1-to-Generic map for an identity relationship

To set up test run	To verify test run
<ol style="list-style-type: none"> <li>1. Set the calling context to EVENT_DELIVERY or ACCESS_REQUEST by selecting the appropriate calling context from the Calling Context combo box.</li> <li>2. Enter the application-specific value in the key of the source business object. This value is unique for the key attribute(s) in Application 1.</li> <li>3. Run the test.</li> </ol>	<ol style="list-style-type: none"> <li>4. Read the resulting generic key value in the destination business object, which has been added to the relationship table for the App1/Generic identity relationship.</li> <li>5. Save the destination business object data in a .bo file (e.g. App1_to_Generic.bo) by selecting the destination business object and selecting Save To from the Context menu.</li> </ol>

2. Testing the outbound map, Generic\_to\_App2, to ensure that the new generic key value is sent to Application 2.

To test an identify relationship in the outbound `Generic_to_App2` map, you must provide the generic key value in your source Test Data. You might want to do either of the following, *but they are both wrong*:

- Put an arbitrary number into the generic business object's primary key attribute, then run the map.
- Create the record directly in the relationship table.

In both cases, Map Designer generates the `RelationshipRuntimeException` or `NullPointerException`. The error occurs because the generic key value has to be in the system for the `SERVICE_CALL_REQUEST` to work properly, and the relationship table is *not* the only place the generic key value is stored.

The correct solution is to first run an inbound `EVENT_DELIVERY` (or `ACCESS_REQUEST`) map that uses the same identity relationship (as described in step 1). Follow the steps in Table 22 to test the outbound `Generic_to_App2` map.

*Table 22. Testing the generic-to-app2 map for an identity relationship*

To set up test run	To verify test run
<ol style="list-style-type: none"> <li>1. Set the calling context to <code>SERVICE_CALL_REQUEST</code> by selecting this calling context from the Calling Context combo box.</li> <li>2. Load the generic business object with the test results from the previous step (e.g. <code>App1_to_Generic.bo</code>).</li> <li>3. Run the test.</li> </ol>	<ol style="list-style-type: none"> <li>4. Read the resulting application-specific key value in the destination business object, which is empty because Application 2 has not generated its key value yet.</li> <li>5. Save the destination business object data in a <code>.bo</code> file (e.g. <code>Generic_to_App2.bo</code>) by selecting the destination business object and selecting <code>Save To</code> from the Context menu.</li> </ol>
<ol style="list-style-type: none"> <li>3. Testing the inbound map, <code>app2_to_generic</code>, to verify that the new key value from Application 2 is associated with the new generic key value. When the calling context is <code>SERVICE_CALL_RESPONSE</code>, an identity relationship must cross-reference the ID in the application-specific business object to the ID in the generic business object. Therefore, for this test, you must specify the generic business object definition. Follow the steps in Table 23.</li> </ol>	

Table 23. Testing the App2\_to\_Generic map for an identity relationship

To set up test run	To verify test run
<ol style="list-style-type: none"><li>1. Set the calling context to SERVICE_CALL_RESPONSE by selecting this calling context from the Calling Context combo box.</li><li>2. Set the generic business object by selecting the name of the appropriate generic business object from the Generic Business Object combo box. Map Designer adds the specified generic business object to the Source Testing Data pane.</li><li>3. Load the application-specific business object with the test results from the previous step (e.g. Generic_to_App2.bo).</li><li>4. In the application-specific business object, enter an application-specific value in the key of the business object.</li><li>5. In the generic business object, enter the generic key value associated with the Application 1 key. This value should be the same key value generated for the generic business object in the EVENT_DELIVERY/ACCESS_REQUEST test (step 1).</li><li>6. Run the test.</li></ol>	<ol style="list-style-type: none"><li>7. Read the resulting generic key value in the destination business object, which should be the same value you entered in the generic source business object.</li><li>8. You can use Relationship Manager to verify that the correct application-specific key values are associated with this generic key value for this identity relationship.</li></ol>

Testing for other verbs involves similar steps. For more detailed information on the actions of relationship methods for an identity relationship, see Chapter 8, “Implementing relationships,” on page 257.

## Testing a lookup relationship

To test point-to-point mapping (from Application 1 to Application 2) for a lookup relationship you use two maps:

- From Application 1’s application-specific business object to a generic business object—App1\_to\_Generic
- From the generic business object to Application 2’s application-specific business object—Generic\_to\_App2

**Example:** Figure 45 shows an example of a point-to-point communication of customer data between a Clarify application and an SAP application. If each application uses a special static code to identify geographic states, these three business objects can be related with a lookup relationship. Therefore, each map includes Custom transformations that do static lookups. For more information, see the “Static Lookup” activity example in “Example 3: Using Static Lookup for conversion” on page 157. As each of these maps executes, these relationship methods access the calling context to determine the actions to take.

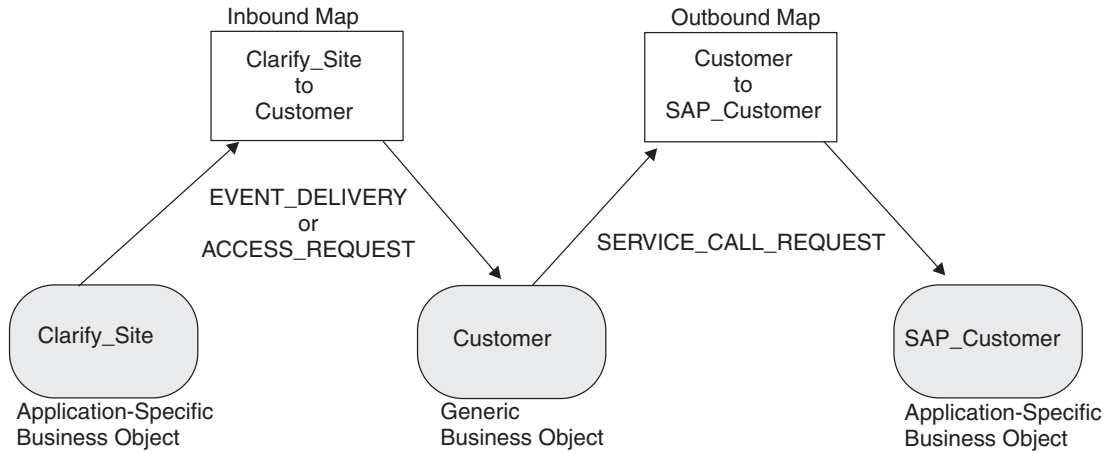


Figure 45. Maps involved in point-to-point testing of a lookup relationship

To test the Create verb, you need to verify that an existing application-specific lookup value in Application 1 (Clarify application in Figure 45) causes the associated generic lookup value to be added to the generic business object *and* the associated application-specific lookup value in Application 2 (SAP application in Figure 45) to be added to its business object. Therefore, testing involves two steps:

1. Testing the inbound map, App1\_to\_Generic, to send in an existing lookup value from Application1 and ensuring that the associated generic lookup value is obtained for the generic business object. Follow the steps in Table 24.

Table 24. Testing the App1-to-Generic map for a lookup relationship

**To set up test run**

1. Set the calling context to EVENT\_DELIVERY or ACCESS\_REQUEST by selecting the appropriate calling context from the Calling Context combo box.
2. Enter the application-specific value in the lookup field of the source business object. This value is an existing lookup value whose data is already loaded in the App1/Generic relationship table.
3. Run the test.

**To verify test run**

4. Read the resulting generic lookup value in the destination business object, which has been obtained to the relationship table for the App1/Generic lookup relationship.
5. Save the business object data in a .bo file (e.g. App1\_to\_Generic.bo) by highlighting the destination business object and selecting Save To from the Context menu.

2. Testing the outbound map, Generic\_to\_App2, to send in the generic lookup value and ensuring that the associated lookup value is obtained for Application 2. Follow the steps in Table 25.

Table 25. Testing the Generic-to-App2 Map for a lookup relationship

**To set up test run**

1. Set the calling context to SERVICE\_CALL\_REQUEST by selecting this calling context from the Calling Context combo box.
2. Load the generic business object with the test results from the previous step (e.g. App1\_to\_Generic.bo).
3. Run the test.

**To verify test run**

4. Read the resulting application-specific key value in the destination business object, which contains the Application 2 lookup value.
5. Save the business object data in a .bo file (e.g. Generic\_to\_App2.bo) by highlighting the destination business object and selecting Save To from the Context menu.

**Note:** A lookup relationship can be tested for the SERVICE\_CALL\_RESPONSE calling context. However, this case usually only is required if the map is doing something else that requires the lookup data. The relationship methods for a lookup relationship in the Mapping API never write data to a relationship table.

---

## Debugging maps

This section provides the following information about debugging a map:

- “Resolving run-time errors”
- “Debugging tips” on page 101

For information on how to test relationships, see “Testing maps that contain relationships” on page 95.

### Resolving run-time errors

Even if your map compiled successfully, you can get a run-time error during the map execution in the Debugger.

**Important:** When resolving run-time errors, make sure that there are *no* pending or failed events related to the dependent business object *before* starting the map debugger.

**Example 1:** You have an outbound map with the generic business object on one side and an application specific business object on the other side. Let us assume that this map has an identity relationship in it.

1. Go to the Test tab and select the calling context SERVICE\_CALL\_REQUEST.
2. Select the verb “Update.”
3. Run the test.

**Result:** An error message like the one below displays:

```
Exception at step 17,  
attribute <attribute name>,  
java.lang.nullpointerexception
```

This exception is happening because the map is trying to update an entry in the repository that is not created in the first place. Ideally, you should ensure that the sequence of steps is correct. You should look at the database for relationship entries pertaining to the map in question. You should then draw the conclusions based on whether it is ready for SERVICE\_CALL\_REQUEST or not.

**Example 2:** You have the following line of the mapping code for Customer.CustomerId:

```
_cw_CpBTBSourceValue = ObjSAP_CustomerMaster.get("CustomerIdd");
```

Clearly, it contains a typo (an extra letter *d* in the name of the attribute). Unfortunately, the compiler does not catch this error because the error is in a string constant. There is no way for the compiler to verify what a “correct” constant value should be. However, when you run the map, the following ICS error dialog displays:

```
ICS Error: Exception at step 3, attribute CustomerId, Exception msg  
number - 11030, Error11030 Attribute CustomerIdd doesn't exist in business  
object SAP_CustomerMaster.
```

When you get this run-time error, leave the Test tab and fix the map.

## Debugging tips

This section provides the following tips for making the debugging of a map easier:

- “Using logging messages”
- “Writing safe mapping code”

**Important:** When debugging a map, make sure that there are *no* pending or failed events related to the dependent business object *before* starting the map debugger.

### Using logging messages

Use the `logInfo()` method for tracking the map execution. It takes a `String` as an argument, which is sent on the InterChange Server log. You need to type it in Activity Editor for the attribute whose execution needs to be tracked. To make sure that the submap is executed, create a custom transformation rule and use the “Log Information” function block to customize the activity or write the code directly.

**Example:** The code can be as simple as the following:

```
logInfo("in submap");
```

Put it on the first line of code of the destination object’s first attribute in the submap.

**Example:** If you need to track the value of the specific attribute `SAP.CustomerName`, use:

```
logInfo(ObjSAP_CustomerMaster.getString("CustomerName"));
```

You might not always want to see this message. If this is the case, change the `DataValidationLevel` property of the map.

To set the `DataValidationLevel`, select the Map Properties option from the Edit menu of Map Designer and change 0 to 1 or a greater number. The settings are as follows:

0	No data validation
1	IBM data validation level
2 or greater	User-defined data validation

To ensure that the `logInfo` message is not displayed, set `DataValidationLevel` to 1. In your code, before calling the `logInfo()` method, check for a data validation level. Here is the code:

```
if (dataValidationLevel > 1)
    logInfo(ObjSAP_CustomerMaster.getString("CustomerName"));
```

This ensures that `logInfo` is executed only if the data validation level is set to a number greater than 1. If you decide to display the message, change the data validation level setting in the Map Properties to 2.

### Writing safe mapping code

If you customize your transformation rule in Activity Editor or write your own mapping code, you are *not* guaranteed that it will work properly during run time. To make sure that the map continues executing when an error occurs and you get a notification of an error, use the “Catch Error” function block in Activity Editor or follow Java’s way of handling exceptions.

**Example:** Put your code inside the try block, for example:

```
try
{
    BusObj temp = new BusObj("SAP_Order");
    // rest of your code
}
```

Then use a catch block to catch whatever exceptions might occur when the code runs:

```
catch (Exception e)
{
    logInfo(e.toString());
}
```

The `logInfo()` method can be used to send system-generated error messages to the InterChange Server log.



---

## Chapter 5. Customizing a map

This chapter provides describes two ways to generate Java code: using Activity Editor to define transformation rules graphically and writing Java code directly.

This chapter covers the following topics:

- “Overview of Activity Editor” on page 103
- “Working with activity definitions” on page 112
- “Exporting Web services into Activity Editor” on page 159
- “Using bidirectional functionality in Activity Editor” on page 162
- “Importing Java packages and other custom code” on page 164
- “Using variables” on page 169
- “More attribute transformation methods” on page 173
- “Reusing map instances” on page 185
- “Handling exceptions” on page 186
- “Creating custom data validation levels” on page 187
- “Understanding map execution contexts” on page 189
- “Mapping child business objects” on page 193
- “More on using submaps” on page 198
- “Executing database queries” on page 203

---

### Overview of Activity Editor

Using Activity Editor, you can specify the flow of activities for a specific transformation rule graphically, without knowing programming or Java code. For each transformation rule in Map Designer, you can display one activity and its subactivities. You can view the associated attribute’s transformation code graphically, modify it, and have the tool generate the corresponding Java code.

You launch Activity Editor directly from Map Designer (see “Starting Activity Editor” on page 103). At startup, Activity Editor communicates with System Manager to discover the set of activities allowed. After you have finished designing the activity for a particular transformation rule, you save the changes in Activity Editor, and they are communicated to Map Designer.

This section covers the following topics to introduce you to Activity Editor:

- “Starting Activity Editor” on page 103
- “Layout of Activity Editor” on page 104
- “Using Activity Editor functionality” on page 108

### Starting Activity Editor

You launch Activity Editor through the transformation rule column of the Table or Diagram tabs of Map Designer. Perform the following steps to do this:

1. Select the attribute you want to work with.
2. Do one of the following:
  - Double-click the attribute’s corresponding cell of the transformation rule column.

- Click the bitmap icon in the corresponding cell of the transformation rule column.

**Result:** Map Designer’ response to these actions depends on the following:

- Whether the code is still in auto-upgrade mode  
Transformation code is in auto-upgrade mode if Map Designer has generated it, and you have not customized it in any way. When you customize auto-upgrade code, Activity Editor displays a confirmation prompt notifying you that saving this code takes it out of auto-update mode. For code not in auto-update mode, Map Designer displays the transformation rule in blue italic font in the transformation rule column.  
If the transformation code is *not* in auto-update mode (that is, you have modified the autogenerated code), Map Designer opens Activity Editor in Java view when you double-click the attribute’s transformation rule cell or click the mapping rule icon.
- The type of transformation defined  
Transformation code that is in auto-update mode is generated from one of the standard transformations that Map Designer provides on the combo box of the transformation rule column. When you double-click the attribute’s transformation rule cell or click the mapping rule icon, the type of transformation determines what Map Designer displays:
  - For the Custom transformation, Map Designer opens Activity Editor on the transformation code.
  - For all other standard transformations (Set Value, Join, Split, Submap, and Cross-Reference), Map Designer displays the transformation’s dialog. Click the View Code push-button on this dialog to open Activity Editor in a new window with the attribute name in the title bar. You can open multiple instances of Activity Editor at the same time.

## Layout of Activity Editor

Activity Editor has two main views: Graphical view and Java view. Depending on the nature of the activity, at any given time, only one view is visible. Thus, if Map Designer invokes Activity Editor to display a graphical activity, Activity Editor will startup with the Graphical view. If you choose to translate this graphical activity into Java code, the Java view will display in place of the Graphical view.

**Restriction:** Once the activity has changed to Java code, it will not be converted back to the graphical nature.

Both views have common Window elements in their Design and Quick view modes, as described in Table 26.

Table 26. Common Window elements

Window element	Description
Title Bar	Contains the name of the application (Activity Editor), application icon, and the main activity’s name.
Menu	Contains the primary menus (Design mode only).
Toolbar	Contains dockable toolbars with shortcuts to various functions and tools (Design mode only).

Table 26. Common Window elements (continued)

Window element	Description
Document Display Area	Displays the representation of the activity definition. It is organized with a workbook look.
Status Bar	Displays status information and some handy shortcuts.

## Working in Graphical view

If Map Designer opens Activity Editor with an activity definition that has a graphical nature, Activity Editor will display the activity definition in Graphical view in one of two available display modes: Design mode or Quick view mode.

- **Design mode:** In Design mode, Activity Editor resembles a regular application--in addition to the main editing window, it has a menu bar, toolbars, and the Library, Content, and Properties windows that support your editing needs during the design stage of the activity definition.

Figure 46 shows the Graphical view in Design mode.

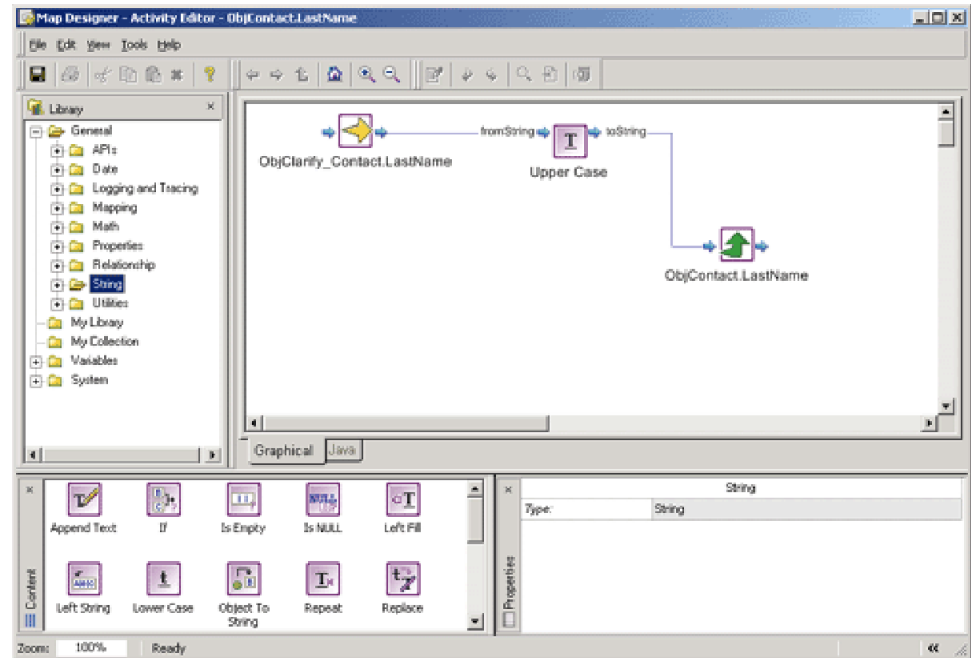


Figure 46. Graphical view in Design mode

The Graphical view has four main windows: the Activity Workbook window, the Library window, the Content window, and the Properties window.

- Activity Workbook window--This window is the main activity editing area, and is usually referred to as the editing canvas. It is also known as the activity canvas or graphical canvas. This area is where you drag and drop the function blocks.
- 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 “Identifying supported function blocks” on page 116), and you can expand them to show the actual function blocks. You can also view the function blocks as icons in the Content window.

In addition, the Library 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.
- My Library--This folder enables you to customize the Library window. It contains any user-defined function blocks that have been specified in the Activity Settings view in System Manager. This folder also contains any Web services function blocks that have been exported from System Manager.
- 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 you can create your own reusable component group. For more information, see “Steps for defining activity group blocks” on page 115.
- Variables--this folder contains global variables accessible to the current activity. It typically contains the port’s business object variables, all of the other business objects and variables defined in the scenario, and the global variable `cwExecCtx`.
- Content window--This window contains a large icon list of the available function blocks under 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 drag-and-drop a function block onto the editing canvas to create part of the activity flow.
- Properties window--This window displays the properties of the currently selected function block in a gridlike layout. Some properties are editable; others are read-only.
- **Quick view mode:** In Quick view mode, Activity Editor only displays the main editing canvas; all other supporting windows (Library, Content, and Properties); the menu bar; and the toolbars are hidden.

Figure 47 shows the Graphical view in Quick view mode.

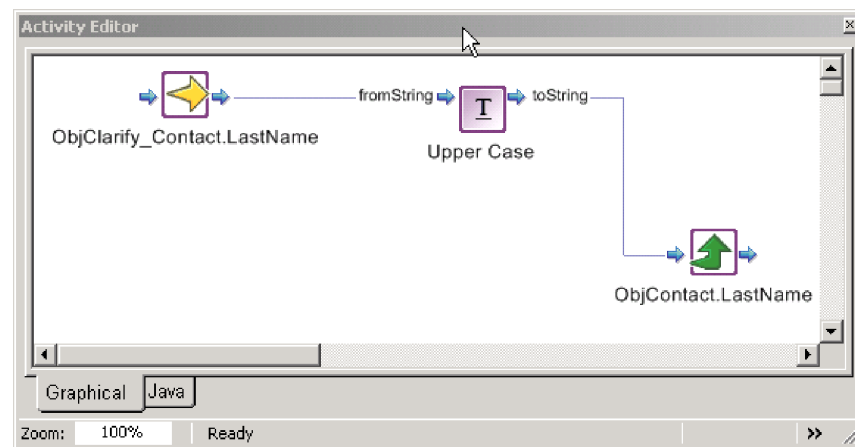


Figure 47. Graphical view in Quick view mode

Initially, when an activity definition that has a graphical nature opens, Activity Editor displays the top-level view of the definition in a tabbed window. Inside the tab window is the *editing canvas*. For information on working with activity definitions on the editing canvas, see “Working with activity definitions” on page 112.

## Working in Java view

If Map Designer opens Activity Editor with an activity definition that contains only custom Java code, Activity Editor displays the activity definition in Java view. Similar to Graphical view, Activity Editor is available in Java view in two display modes: Design mode and Quick view mode.

- **Design mode:** In Design mode, the Java view of Activity Editor contains the main Java WordPad for viewing and editing custom Java code to provide the definition for the activity. The WordPad is contained in a tabbed window area. In addition to the regular editing options in a WordPad (Cut, Copy, Paste, Delete, Select All, Undo, Redo), the Java WordPad provides syntax highlighting for the Java Programming language.

By default, comments are green, string literals are pink, and keywords are blue.

**Tip:** You can customize the syntax highlighting schemes in the Preferences dialog.

Figure 48 shows the Java view in Design mode.

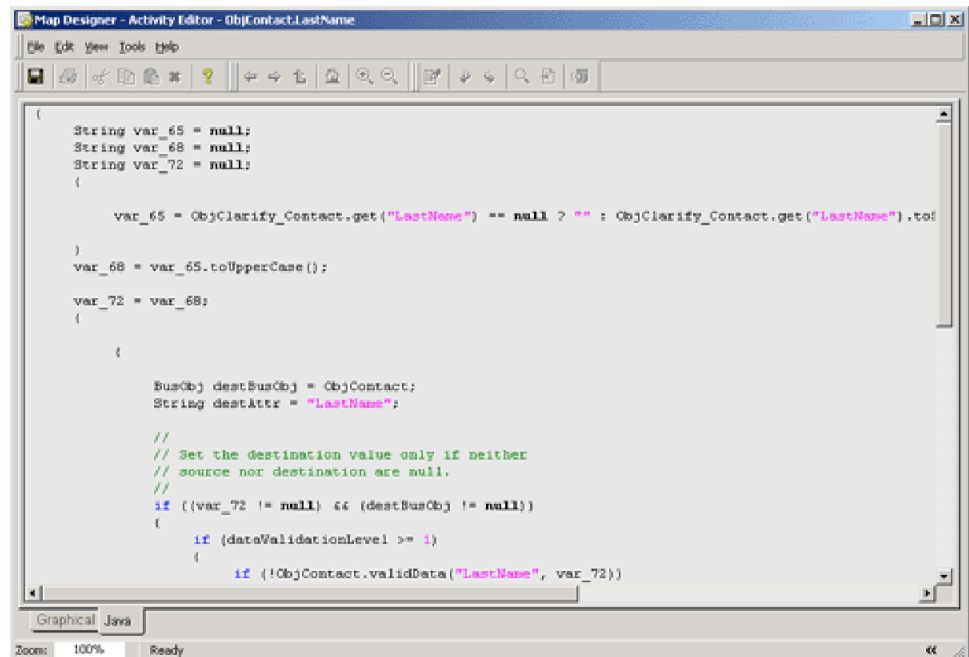


Figure 48. Java view in Design mode

- **Quick view mode:** In Quick view mode, the Java view only displays the WordPad. Figure 49 shows the Java view in Quick view mode.

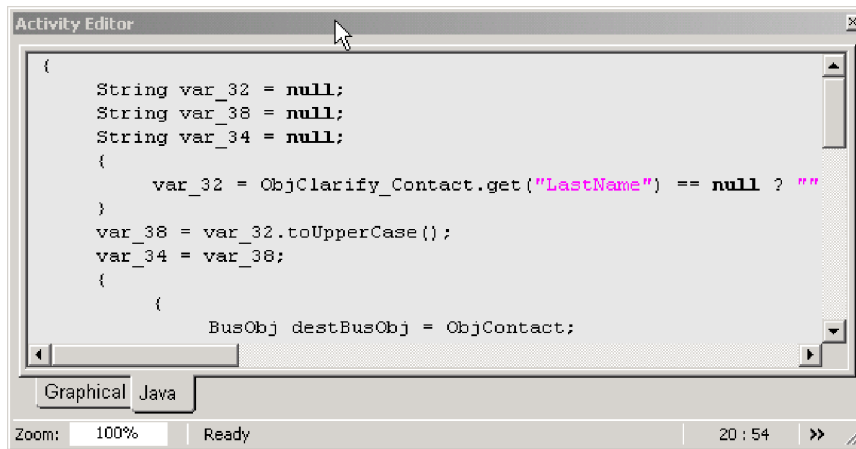


Figure 49. Java view in Quick view mode

**Tip:** To change from Quick view mode to Design mode, click the >> button on the status bar. If you do not see the >> button, resize the Quick view window horizontally until the button appears.

**Note:** Initially, the Java view will be in read-only mode. To enter customized Java code, click the Edit Code toolbar button, or select Edit Code from the Tools menu.

## Using Activity Editor functionality

You can access Activity Editor's functionality using any of the following:

- Pull-down menus
- Context menu
- Toolbar buttons
- Keyboard shortcuts

### Activity Editor pull-down menus and keyboard shortcuts

Activity Editor provides the following pull-down menus:

- File menu
- Edit menu
- View menu
- Tools menu
- Help menu

The following sections describe the options of each of these menus and their associated keyboard shortcuts.

**Functions of the File menu:** The File pull-down menu of Activity Editor provides the following options:

- Save [Ctrl+S]--Saves the activity to Map Designer.
- Print Setup [Ctrl+Shift+P]--Opens the Print Setup dialog box for specifying print options.
- Print Preview--Switches Activity Editor to print preview mode.
- Print [Ctrl+P]--Opens the Print dialog box for printing the current activity.
- Close --Closes Activity Editor.

**Functions of the Edit menu:** The Edit pull-down menu of Activity Editor provides the following options:

- Undo [Ctrl+A]--Clears the last change you made and restores the previous version.
- Redo [Ctrl+Y]--Restores a change that was previously removed with the Undo command.
- Cut [Ctrl+X]--Deletes the selected item and copies it to the clipboard.
- Copy [Ctrl+C]--Copies the selected item to the clipboard.
- Paste [Ctrl+P]--Pastes the object in the clipboard to the cursor position if they are compatible.
- Delete [Del]--Deletes the selected item.
- Select All [Ctrl+A]--Selects all items.
- Find [Ctrl+F]--Finds the specific text in the editing area.
- Replace [Ctrl+H]--Replaces specific text with different text in the editing area.
- Goto Line [Ctrl+G]--Moves the cursor to a specific line.

**Functions of the View menu:** The View pull-down menu of Activity Editor provides the following options:

- Design mode--Toggles between Design mode and Quick view mode. (Only one mode is enabled at a single time.)
- Quick view mode--Toggles between Quick view mode and Design mode. (Only one mode is enabled at a single time.)
- Go To--Provides the following options:
  - Back [Alt+Left Arrow]--Moves backward in the navigation history in the Graphical view.
  - Forward [Alt+Right Arrow]--Moves forward in the navigation history in the Graphical view.
  - Up One Level--Displays the diagram from one higher level.
  - Home [Alt+Home]--Goes to the top-level diagram in Graphical view.
- Zoom In [Ctrl++]--Magnifies content in Activity Editor.
- Zoom Out [Ctrl+-]--Minimizes content in Activity Editor.
- Zoom To [Ctrl+M]--Opens the Zoom dialog box for specifying a particular zoom level.
- 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 a submenu for displaying toolbars (Standard, Graphics, and Java) that toggle on and off.
- Status Bar--Toggles the status bar on and off.
- Preferences... [Ctrl+U]--Opens the Preferences dialog box for specifying the default behavior of Activity Editor.

**Functions of the Tools menu:** The Tools pull-down menu of Activity Editor provides the following option:

- Translate [Ctrl+T]--Translates the current activity to Java code and opens the Java view.
- Edit Code--Allows you to edit code in Java.
- Check for Unmatched Delimiters--Checks for unmatched delimiters in the Java code.

- Expression Builder--Opens the Expression Builder utility.

**Functions of the Help menu:** The Help pull-down menu of Activity Editor provides the following options:

- Help Topics [F1]--Opens the context-sensitive Help topics
- Documentation--Opens the InterChange Server documentation.

### Context menu

Activity also provides a context menu for performing many tasks on the editing canvas. You access the Context menu by right-clicking the editing canvas. The Context menu provides the following options:

- New Constant--Creates a new Constant container 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 editing canvas.
- Add To do--Creates a new reminder component in the activity.
- Add To My Collection--Creates a new group component for reuse in the Library window.

### Activity Editor toolbars

Activity Editor provides three toolbars for common tasks you need to perform.

- Standard toolbar
- Graphics toolbar
- Java toolbar

The functions of the toolbar buttons are the same as their corresponding menu items.

**Tip:** To identify the function of each toolbar button, roll over each button with your mouse cursor.

**Standard toolbar:** Figure 50 shows the Standard toolbar.

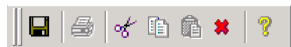


Figure 50. Activity Editor Standard toolbar

Table 27 provides the function of each Standard toolbar button (left to right) and the corresponding menu command.

Table 27. Functions of Standard toolbar buttons

Function	Corresponding menu command
Save Activity	File > Save
Print Activity	File > Print
Cut	Edit > Cut
Copy	Edit > Copy
Paste	Edit > Paste
Delete	Edit > Delete
Help	Help > Help Topics



**Graphics toolbar:** Figure 51 shows the Graphics toolbar.

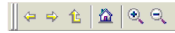


Figure 51. Activity Editor Graphics toolbar

Table 28 provides the function of each Graphics toolbar button (left to right) and the corresponding menu command.

Table 28. Functions of Graphics toolbar buttons

Function	Corresponding menu command
Back	View > Go To > Back
Forward	View > Go To > Forward
Up One Level	View > Go To > Up One Level
Home	View > Go To > Home
Zoom In	View > Zoom In
Zoom Out	View > Zoom Out

Figure 52 shows the Java toolbar.



Figure 52. Activity Editor Java toolbar

Table 29 provides the function of each Java toolbar button (left to right) and the corresponding menu command.

Table 29. Functions of Java toolbar buttons

Function	Corresponding menu command
Edit Code	Tools > Edit Code
Undo	Edit > Undo
Redo	Edit > Redo
Find Text	Edit > Find
Goto Line	Edit > Goto Line
Expression Builder	Tools > Expression Builder

Status bar elements: Activity Editor also provides a Status bar, as shown in Figure 53.



Figure 53. Activity Editor Status bar

Table 30 describes the functionality of each Status bar element, left to right.

Table 30. Functions of Status bar elements

Element	Function
Zoom: 100%	Edit box for specifying a zooming percentage
Ready	Status message
10.9	Navigation pane showing the current position of the I-bar in the Java editor

Table 30. Functions of Status bar elements (continued)

Element	Function
>> (Shown in Quick view mode)	Toggle between Design mode and quick view mode
<< (Shown in Design mode)	

---

## Working with activity definitions

Activity Editor is used to define and modify activity definitions for transformation rules. This is done on the editing canvas using the canvas components: function blocks, connection links, tags, and New Constant icon.

### Using function blocks

An activity definition is built with *function blocks*, which represent discrete parts of the activity definition, such as a constant, a variable or a programming method. Many of the function blocks in Activity Editor correspond to individual methods in the Mapping API.

You place function blocks on the editing canvas by dragging and dropping them from either the Library or Content window. Once you drop a function block on the editing canvas, you can move it around, by clicking it to select it and dragging 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 you drop the function block on the editing canvas, its input and output ports are represented by arrows. 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).

For a description of supported function blocks in the Map Designer and Relationship Designer contexts, see “Identifying supported function blocks” on page 116.

**Note:** In addition to the standard function blocks that Activity Editor provides, you can export Web services from System Manager into Activity Editor. The export process converts each method in the Web service to a function block, which you can then use in activity definitions the same way as other function blocks. For more information, see “Exporting Web services into Activity Editor” on page 159.

You can also import your own Java library for use as function blocks in 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 Activity Editor. For more information, see “Importing Java packages and other custom code” on page 164.

### Tip: Using function blocks directly in Map Designer

If you only want to use one standard function block in a custom transformation, you can configure the function block in the Preferences dialog for direct use in Map Designer. Then after selecting the source and destination attributes for the Custom transformation, you can select the configured function block in the transformation rule combo box under Custom in Map Designer.

**Steps for using function blocks directly in Map Designer:** Perform the following steps to set up direct use of function blocks in Map Designer:

1. Start Map Designer. For information on starting Map Designer, see “Starting Map Designer” on page 15
2. From the View menu, select Preferences, or use the keyboard shortcut of Ctrl+U.

**Result:** The Preferences dialog opens.

3. In the Preferences dialog, select the Custom Mapping tab.
4. From the list of standard function blocks, select the function blocks to be used directly in Map Designer.

Figure 54 shows the Custom Mapping tab with the selected function blocks.

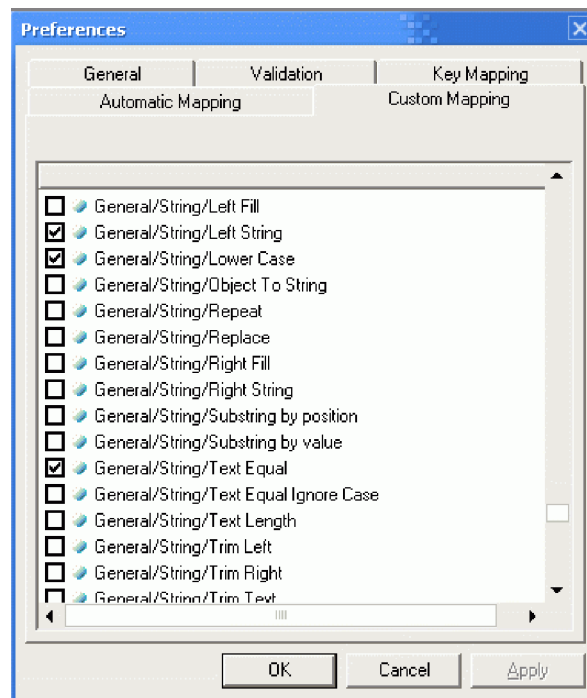


Figure 54. Preferences screen with Custom Mapping tab showing

5. Click OK.

**Result:** The configured function blocks will be available for direct use in the transformation rule combo box under Custom in Map Designer.

## Using connection links

Function blocks are connected by *connection links*. The 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:** Outgoing ports can connect to multiple connection links, but incoming ports can only connect to one connection link.

**Tip:** When you drag-and-drop to connect function blocks together, Activity Editor uses the option set in the Validation tab of the Preferences dialog to determine if it needs to validate and check whether the "from" parameter type is the same as the "to" parameter type.

- By default, this preference is set to "Warning," meaning that when you create a link between two parameters that are of different types, Activity Editor will show a message warning you that this may lead to a compile error.
- Setting the option to "Ignore" tells Activity Editor not to do any validation.
- Setting the option to "Error" tells Activity Editor not to allow you to create links between different types.

**Example:** To specify that the output of function block A should go to the input of function block B, perform the following steps:

1. Click and hold down the left mouse button on the outgoing port of function block A.
2. While continuing to hold down the left mouse button, move the cursor onto the incoming port of function block B.
3. Release the left mouse button.

**Result:** The connection link is placed between function block A's out-port and function block B's in-port. Graphically, the connection link will appear as a right-angled line between components. If function block B's in-port is already connected with another connection link, the newer connection link will replace the existing connection link.

## Using label, description, comment, and to-do tags

The System folder (located in the Library and Content windows) contains function blocks for adding *label*, *description*, *comment*, and *to-do* tags to the activity definitions. These tags help identify each activity or subactivity, or serve as a reminder of something that needs to 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. If you want to start a new line, press enter.

To resize a tag, left-click the lower right-hand corner of the tag and hold down the left mouse button while dragging the tag to the desired size.

Figure 55 shows resizing a label tag and entering multiple lines of text.

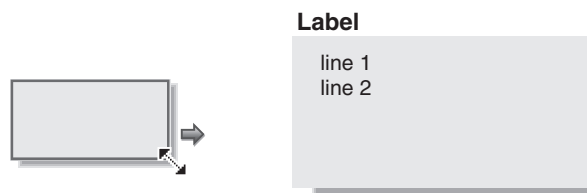


Figure 55. Resizing a label and entering multiple lines of text

**Restriction:** Each of these editing components has a minimize size, so the components cannot be resized to be smaller than a certain size.

To move the tag around the canvas, click the edge of the component and drag-and-drop it.

## Using the New Constant function block

Activity Editor has a *New Constant function block* 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 window and Content window. Activity Editor displays a text edit box on top of the function block icon for you to enter the value of the Constant. To revise this value, double-click the Constant icon and enter the new value. Constants contain one outgoing port.

**Note:** The Constant is the only activity definition component that accepts only a single line for the value. This is because the constant is translated to a Java code String, and the system cannot translate a multi-line constant value. If multi-line input is required, use the "\n" value to separate between lines in the Constant.

**Example:** The value "line1\nline2" will tell the system to output the text in two lines.

## Steps for defining activity group blocks

Once you have defined an activity flow with a set of function blocks on the editing canvas, you can select and save it as a named activity group for later reuse in another activity definition. The saved activity group is represented by an icon. The following procedure describes the steps to take.

**Before you begin:** You need to enable "Show child functions in Library window" in the Preference dialog to display the added group.

Perform the following steps:

1. Select the activity components you want to group together on the editing canvas. To select multiple components, hold down the Ctrl key and click each component.
2. Right-click the editing canvas to open the Context menu. Then select Add to My Collection. Alternatively, right-click the component and select Add to My Collection.

**Result:** The Add to My Collection dialog box is displayed.

3. In the Add to My Collection dialog, type a name and a description for the activity group block; and select an icon to represent this group. Then click OK.

**Result:** The activity group block 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.

**Note:** Any input or output parameters that are not connected when the user group is saved will appear as the input/output of this activity group.

**Example:** Figure 56 shows an activity in which the graphical components enclosed in the box are saved as an activity group.

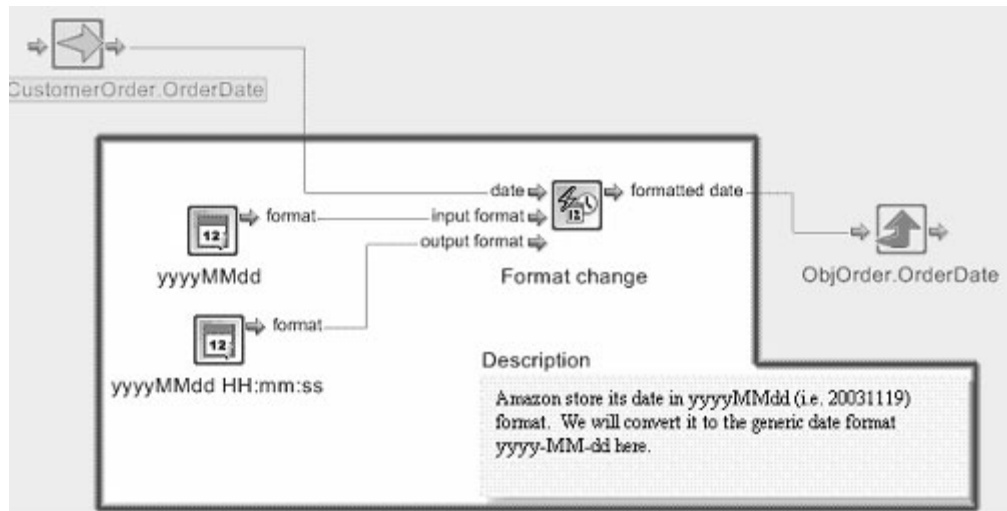


Figure 56. Graphical components that are saved as a single activity

When this activity group is reused, it has an icon representation, as shown in Figure 57.

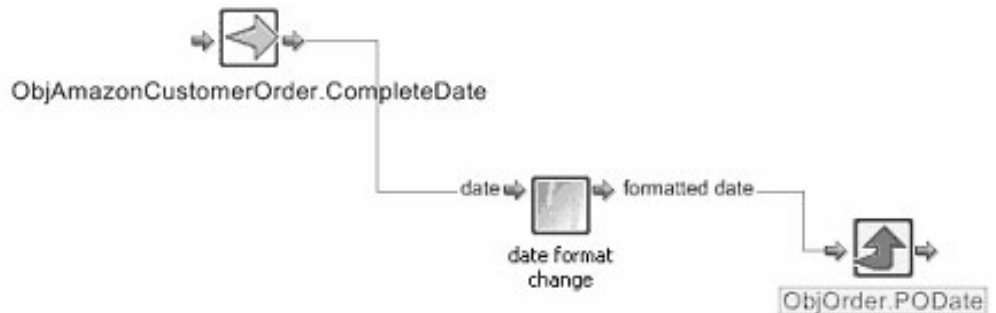


Figure 57. Reused activity group represented as an icon

**Tip:** Double-clicking on this group icon will show the group definition in detail.

## Identifying supported function blocks

The supported function blocks, in the Map Designer context, are organized into the categories shown in the following table. These categories correspond to folders in the Library window and the Content window.

Table 31. Organization of function blocks

Function block folder	Description	For more information
General/APIs/Business Object	Function blocks for working with business objects.	Table 32 on page 118
General/APIs/Business Object/Array	Function blocks for working with Java arrays in the BusObj class.	Table 33 on page 121

Table 31. Organization of function blocks (continued)

Function block folder	Description	For more information
General/APIs/Business Object/Constants	Function blocks for working with Java constants in the BusObj class.	Table 34 on page 122
General/APIs/Business Object Array	Function blocks for working with business object arrays.	Table 35 on page 122
General/APIs/Database Connection	Function blocks for creating and maintaining a database connection.	Table 36 on page 123
General/APIs/Identity Relationship	Function blocks for working with identity relationships.	Table 37 on page 125
General/APIs/Maps	Function blocks for querying and setting run-time values needed for map execution.	Table 38 on page 127
General/APIs\Maps/Constants	Function block constants.	Table 39 on page 127
General/APIs/Maps/Exception	Function blocks for creating new exception objects in a map.	Table 40 on page 128
General/APIs/Participant	Function blocks for setting and retrieving values for participants in identity relationships.	Table 41 on page 129
General/APIs/Participant/Array	Function blocks for creating and working with participant arrays.	Table 42 on page 131
General/APIs/Participant/Constants	Function block constants for use with participants.	Table 43 on page 131
General/APIs/Relationship	Function blocks for manipulating run-time instances of relationships.	Table 44 on page 131
General/Date	Function blocks for working with dates.	Table 45 on page 134
General/Date/Formats	Function blocks for specifying different date formats.	Table 46 on page 135
General/Logging and Tracing	Function blocks for handling log and trace messages.	Table 47 on page 135
General/Logging and Tracing/Log Error	Function blocks for formatting error messages.	Table 48 on page 136
General/Logging and Tracing/Log Information	Function blocks for formatting informational messages.	Table 49 on page 136
General/Logging and Tracing/Log Warning	Function blocks for formatting warning messages.	Table 50 on page 137
General/Logging and Tracing/Trace	Function blocks for formatting trace messages.	Table 51 on page 137
General/Mapping	Function blocks for executing maps within a specified context.	Table 52 on page 138
General/Math	Function blocks for basic mathematical tasks.	Table 53 on page 138

Table 31. Organization of function blocks (continued)

Function block folder	Description	For more information
General/Properties	Function blocks for retrieving configuration property values.	Table 54 on page 140
General/Relationship	Function blocks for maintaining and querying identity relationships.	Table 55 on page 141
General/String	Function blocks for manipulating String objects.	Table 56 on page 141
General/Utilities	Function blocks for throwing and catching exceptions, as well as looping, moving attributes, and setting conditions.	Table 57 on page 143
General\Utilities/Vector	Function blocks for working with Vector objects.	Table 58 on page 144

The following tables describe the function blocks in each category, including the acceptable values for their inputs and outputs.

Table 32. General/APIs/Business Object

Name	Description	Inputs and outputs with acceptable values
Copy	Copies all attribute values from the input business object. API: BusObj.copy()	Inputs: <ul style="list-style-type: none"> <li>copy to--BusObj</li> <li>copy from--BusObj</li> </ul>
Duplicate	Creates a business object exactly like the original one. API: BusObj.duplicate()	Inputs: original--BusObj Outputs: duplicate--BusObj
Equal Keys	Compares business object 1's and business object 2's values, to determine whether they are equal. API: BusObj.equalKeys()	Inputs: <ul style="list-style-type: none"> <li>business object 1--BusObj</li> <li>business object 2--BusObj</li> </ul> Outputs: key values equal?-- boolean
Equals	Compares business object 1's and business object 2's values, including child business objects, to determine whether they are equal. API: BusObj.equals()	Inputs: <ul style="list-style-type: none"> <li>business object 1--BusObj</li> <li>business object 2--BusObj</li> </ul> Outputs: equal?-- boolean
Exists	Checks for the existence of a business object attribute with a specified name. API: BusObj.exists()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: exists?-- boolean
Get Boolean	Retrieves the value of a single attribute, as a boolean, from a business object. API: BusObj.getBoolean()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value-- boolean



Table 32. General/APIs/Business Object (continued)

Name	Description	Inputs and outputs with acceptable values
Get Business Object	Retrieves the value of a single attribute, as a BusObj, from a business object.  API: BusObj.getBusObj()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--BusObj
Get Business Object Array	Retrieves the value of a single attribute, as a BusObj Array, from a business object.  API: BusObj.getBusObjArray()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--BusObjArray
Get Business Object Type	Retrieves the name of the business object definition on which this business object was based.  API: BusObj.getType()	Inputs: business object--BusObj  Outputs: type--String
Get Double	Retrieves the value of a single attribute, as a double, from a business object.  API: BusObj.getDouble()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--double
Get Float	Retrieves the value of a single attribute, as a float, from a business object.  API: BusObj.getFloat()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--float
Get Int	Retrieves the value of a single attribute, as an integer, from a business object.  API: BusObj.getInt()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--int
Get Long	Retrieves the value of a single attribute, as a long, from a business object.  API: BusObj.getLong()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--long
Get Long Text	Retrieves the value of a single attribute, as a long text, from a business object.  API: BusObj.getLongText()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--String
Get Object	Retrieves the value of a single attribute, as an object, from a business object. The attribute can be specified as either the attribute name or the attribute position.  API: BusObj.get()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String, int</li> </ul> Outputs: value--Object
Get String	Retrieves the value of a single attribute, as a string, from a business object.  API: BusObj.getString()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> </ul> Outputs: value--String

Table 32. General/APIs/Business Object (continued)

Name	Description	Inputs and outputs with acceptable values
Get Verb	Retrieves this business object's verb.  API: BusObj.getVerb()	Inputs: business object--BusObj  Outputs: verb--String
Is Blank	Finds out whether the value of an attribute is set to a zero-length string.  API: BusObj.isBlank()	Inputs: • business object--BusObj • attribute--String  Outputs: blank?--boolean
Is Business Object	Tests whether the value is a business object (BusObj).	Inputs: value--Object  Outputs: result--boolean
Is Key	Finds out whether a business object's attribute is defined as a key attribute.  API: BusObj.isKey()	Inputs: • business object--BusObj • attribute--String  Outputs: key?--boolean
Is Null	Finds out whether the value of a business object's attribute is null.  API: BusObj.isNull()	Inputs: • business object--BusObj • attribute--String  Outputs: null?--boolean
Is Required	Finds out whether a business object's attribute is defined as a required attribute.  API: BusObj.isRequired()	Inputs: • business object--BusObj • attribute--String  Outputs: required?--boolean
Iterate Children	Iterates through the child business object array.	Inputs: • business object--BusObj • attribute--String • current index--int • current element--BusObj
Key to String	Retrieves the values of a business object's primary key attributes as a string.  API: BusObj.keysToString()	Inputs: business object--BusObj  Outputs: key string--String
New Business Object	Creates a new business object instance (BusObj) of the specified type.  API: Collaboration.BusObj()	Inputs: type--String  Outputs: business object--BusObj
Set Content	Sets the contents of this business object to another business object. The two business objects will own the content together. Changes made to one business object will be reflected in the other business object.  API: BusObj.setContent()	Inputs: • business object--BusObj • content--BusObj
Set Default Attribute Values	Sets all attributes to their default values.  API: BusObj.setDefaultAttrValues()	Inputs: business object--BusObj

Table 32. General/APIs/Business Object (continued)

Name	Description	Inputs and outputs with acceptable values
Set Keys	Sets the values of the "to" business object's key attributes to the values of the key attributes in "from" business object.  API: BusObj.setKeys()	Inputs: <ul style="list-style-type: none"> <li>from business object--BusObj</li> <li>to business object--BusObj</li> </ul>
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.  API: BusObj.setWithCreate()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> <li>value--BusObj, BusObjArray, Object</li> </ul>
Set Verb	Sets the verb of a business object.  API: BusObj.setVerb()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>verb--String</li> </ul>
Set Verb with Create	Sets the verb of a child business object, creating the child business object if one does not already exist.  API: BusObj.setVerbWithCreate()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> <li>verb--String</li> </ul>
Set Value	Sets a business object's attribute to a specified value of a particular data type.  API: BusObj.set()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> <li>value--boolean, double, float, int, long, Object, String, BusObj</li> </ul>
Shallow Equals	Compares business object 1 and business object 2's values, excluding child business objects, to determine whether they are equal.  API: BusObj.equalsShallow()	Inputs: <ul style="list-style-type: none"> <li>business object 1--BusObj</li> <li>business object 2--BusObj</li> </ul> Outputs: equal?--boolean
To String	Gets the values of all attributes in a business object as string.  API: BusObj.toString()	Inputs: business object--BusObj  Outputs: string--String
Valid Data	Checks whether the specified value is a valid type for a specified attribute.  API: BusObj.validData()	Inputs: <ul style="list-style-type: none"> <li>business object--BusObj</li> <li>attribute--String</li> <li>value--Object, BusObj, BusObjArray, String, long, int, double, float, boolean</li> </ul> Outputs: valid?--boolean

Table 33. General/APIs/Business Object/Array

Name	Description	Inputs and outputs with acceptable values
Get BusObj At	Retrieves the element at the specified index in the business object array.	Inputs: <ul style="list-style-type: none"> <li>array--BusObj[]</li> <li>index--int</li> </ul> Outputs: business object--BusObj
New Business Object Array	Creates a new business object array.	Inputs: size--int  Outputs: array--BusObj[]

Table 33. General/APIs/Business Object/Array (continued)

Name	Description	Inputs and outputs with acceptable values
Set BusObj At	Sets the element at the specified index in the business object array.	Inputs: <ul style="list-style-type: none"> <li>array--BusObj[]</li> <li>index--int</li> <li>business object--BusObj</li> </ul>
Size	Retrieves the size of the business object array	Inputs: array--BusObj[]  Outputs: size--int

Table 34. General/APIs/Business Object/Constants

Name	Description	Inputs and outputs with acceptable values
Verb: Create	Business object verb "Create".	Outputs: Create--String
Verb: Delete	Business object verb "Delete".	Outputs: Delete--String
Verb: Retrieve	Business object verb "Retrieve".	Outputs: Retrieve--String
Verb: Update	Business object verb "Update".	Outputs: Update--String

Table 35. General/APIs/Business Object Array

Name	Description	Inputs and outputs with acceptable values
Add Element	Adds a business object to this business object  API: BusObjArray.addElement()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>element--BusObj</li> </ul>
Duplicate	Creates a business object array exactly like the original one.  API: BusObjArray.duplicate()	Inputs: original--BusObjArray  Outputs: duplicate--BusObjArray
Equals	Compares business object array 1's and business object array 2's values, to determine whether they are equal.  API: BusObjArray.equals()	Inputs: <ul style="list-style-type: none"> <li>array 1--BusObjArray</li> <li>array 2--BusObjArray</li> </ul> Outputs: equal?-- boolean
Get Element At	Retrieves a single business object by specifying its position in the business object array.  API: BusObjArray.elementAt()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>index--int</li> </ul> Outputs: element--BusObj
Get Elements	Retrieves the contents of this business object array.  API: BusObjArray.getElements()	Inputs: business object array--BusObjArray  Outputs: element--BusObj[]
Get Last Index	Retrieves the last available index from a business object array.  API: BusObjArray.getLast Index()	Inputs: business object array--BusObjArray  Outputs: last index--int
Is Business Object Array	Tests whether value is a business object array (BusObjArray).	Inputs: value--Object  Outputs: result--boolean

Table 35. General/APIs/Business Object Array (continued)

Name	Description	Inputs and outputs with acceptable values
Max attribute value	Retrieves the maximum values for the specified attribute among all elements in this business object array.  API: BusObjArray.max()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>attribute--String</li> </ul> Outputs: max--String
Min attribute value	Retrieves the minimum value for the specified attribute among all elements in this business object array.  API: BusObjArray.min()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>attribute--String</li> </ul> Outputs: min--String
Remove All Elements	Removes all elements from the business object array.  API: BusObjArray.removeAllElements()	Inputs: business object array--BusObjArray
Remove Element	Removes a business object element from a business object array.  API: BusObjArray.removeElement()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>element--BusObj</li> </ul>
Remove Element At	Removes an element at a particular position in this business object array.  API: BusObjArray.removeElementAt()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>index--int</li> </ul>
Set Element At	Sets the value of a business object in the business object array.  API: BusObjArray.setElementAt()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>index--int</li> <li>element--BusObj</li> </ul>
Size	Gets the number of elements in this business object array.  API: BusObjArray.size()	Inputs: business object array--BusObjArray  Outputs: size--int
Sum	Adds the values of the specified attribute for all business objects in this business object array.  API: BusObjArray.sum()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>attribute--String</li> </ul> Outputs: sum--double
Swap	Reverses the positions of two business objects in this business object array.  API: BusObjArray.swap()	Inputs: <ul style="list-style-type: none"> <li>business object array--BusObjArray</li> <li>index 1--int</li> <li>index 2--int</li> </ul>
To String	Retrieves the values in this business object array as a single string.  API: BusObjArray.toString()	Inputs: business object array--BusObjArray  Outputs: string--String

Table 36. General/APIs/Database Connection

Name	Description	Inputs and outputs with acceptable values
Begin Transaction	Begins an explicit transaction for the current connection.  API: CwDBCConnection.beginTransaction()	Inputs: database connection--CwDBCConnection

Table 36. General/APIs/Database Connection (continued)

Name	Description	Inputs and outputs with acceptable values
Commit	Commits the active transaction associated with the current connection.  API: CwDBConnection.commit()	Inputs: database connection--CwDBConnection
Execute Prepared SQL	Executes a prepared SQL Query by specifying its syntax.  API: CwDBConnection.executePreparedSQL()	Inputs: • database connection--CwDBConnection • query--String  Outputs: equal?-- boolean
Execute Prepared SQL with Parameter	Executes a prepared SQL query by specifying its syntax with the specified parameters.  API:CwDBConnection.executePreparedSQL()	Inputs: • database connection--CwDBConnection • query--String • parameters--java.util.Vector
Execute SQL	Executes a static SQL query by specifying its syntax.  API: CwDBConnection.executeSQL()	Inputs: • database connection--CwDBConnection • query--String
Execute SQL with Parameter	Executes a static SQL query by specifying its syntax with the specified parameters.  API: CwDBConnection.executeSQL()	Inputs: • database connection--CwDBConnection • query--String • parameters--java.util.Vector
Execute Stored Procedure	Executes an SQL stored procedure by specifying its name and parameter array.  API: CwDBConnection.executeStored Procedure()	Inputs: • database connection--CwDBConnection • query--String • parameters--java.util.Vector
Get Database Connection	Establishes a connection to a database and returns a CwDBConnection() object.  API: BaseDLM.getDBConnection() or BaseCollaboration.getDBConnection()	Inputs: connection pool name--String  Outputs: database connection--CwDBConnection
Get Database Connection with Transaction	Establishes a connection to a database and returns a CwDBConnection() object.  API: BaseDLM.getDBConnection() or BaseCollaboration.getDBConnection()	Inputs: • connection pool name--String • implicit transaction--boolean  Outputs: database connection--CwDBConnection
Get Next Row	Gets the next row from the query result.  API: CwDBConnection.nextRow()	Inputs: database connection--CwDBConnection  Outputs: row--java.util.Vector
Get Update Count	Gets the number of rows affected by the last write operation to the database.  API: CwDBConnection.getUpdateCount()	Inputs: database connection--CwDBConnection  Outputs: count--int
Has More Rows	Determines whether the query result has more rows to process.  API: CwDBConnection.hasMoreRows()	Inputs: database connection--CwDBConnection  Outputs: more rows?--boolean

Table 36. General/APIs/Database Connection (continued)

Name	Description	Inputs and outputs with acceptable values
In Transaction	Determines whether a transaction is in progress in the current connection.  API: CwDBConnection.inTransaction()	Inputs: database connection--CwDBConnection  Outputs: in transaction?--boolean
Is Active	Determines whether the current connection is active.  API: CwDBConnection.isActive()	Inputs: database connection--CwDBConnection  Outputs: is active?--boolean
Release	Releases use of the current connection, returning it to its connection pool.  API: CwDBConnection.release()	Inputs: database connection--CwDBConnection
Roll Back	Rolls back the active transaction associated with the current connection.  API: CwDBConnection.rollback()	Inputs: database connection--CwDBConnection

Table 37. General/APIs/Identity Relationship

Name	Description	Inputs and outputs with acceptable values
Add My Children	Adds the specified child instances to a parent/child relationship for an identity relationship.  API: IdentityRelationship.addMyChildren()	Inputs: <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• parentChildRelDefName--String</li> <li>• parentParticipantDefName--String</li> <li>• parentBusObj--BusObj</li> <li>• childParticipantDefName--String</li> <li>• childBusObjList--BusObj, BusObjArray</li> </ul>
Delete All My Children	Removes all child instances from a parent/child relationship for an identity relationship belonging to the specified parent.  API: IdentityRelationship.deleteMyChildren()	Inputs: <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• parentChildRelDefName--String</li> <li>• parentParticipantDefName--String</li> <li>• parentBusObj--BusObj</li> <li>• childParticipantDefName--String</li> </ul>
Delete My Children	Removes the specified child instances from a parent/child relationship for an identity relationship belonging to the specified parent.  API: IdentityRelationship.deleteMyChildren()	Inputs: <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• parentChildRelDefName--String</li> <li>• parentParticipantDefName--String</li> <li>• parentBusObj--BusObj</li> <li>• childParticipantDefName--String</li> <li>• childBusObjList--BusObj, BusObjArray</li> </ul>

Table 37. General/APIs/Identity Relationship (continued)

Name	Description	Inputs and outputs with acceptable values
Foreign Key Cross-Reference	<p>Performs a lookup in the relationship table in the relationship database based on the foreign key of the source business object, adding a new relationship instance in the foreign relationship table if the foreign key does not exist.</p> <p>API: IdentityRelationship.foreignKeyXref()</p>	<p>Inputs:</p> <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• relDefName--String</li> <li>• appParticipantDefName--String</li> <li>• genParticipantDefName--String</li> <li>• appSpecificBusObj--BusObj</li> <li>• appForeignAttr--String</li> <li>• genericBusObj--BusObj</li> <li>• genForeignAttr--String</li> </ul>
Foreign Key Lookup	<p>Performs a lookup in a foreign relationship table based on the foreign key of the source business object, failing to find a relationship instance if the foreign key does not exist in the foreign relationship table.</p> <p>API: IdentityRelationship.foreignKeyLookup()</p>	<p>Inputs:</p> <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• relDefName--String</li> <li>• appParticipantDefName--String</li> <li>• appSpecificBusObj--BusObj</li> <li>• appForeignAttr--String</li> <li>• genericBusObj--BusObj</li> <li>• genForeignAttr--String</li> </ul>
Maintain Child Verb	<p>Sets the child business object verb based on the map execution context and the verb of the parent business object.</p> <p>API: IdentityRelationship.maintainChildVerb()</p>	<p>Inputs:</p> <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• relDefName--String</li> <li>• appSpecificParticipantName--String</li> <li>• genericParticipantName--String</li> <li>• appSpecificObj--BusObj</li> <li>• appSpecificChildObj--String</li> <li>• genericObj--BusObj</li> <li>• genericChildObj--String</li> <li>• to_Retrieve--boolean</li> <li>• Is_Composite--boolean</li> </ul>
Maintain Composite Relationship	<p>Maintains a composite identity relationship from within the parent map.</p> <p>API: IdentityRelationship.maintainCompositeRelationship()</p>	<p>Inputs:</p> <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• relDefName--String</li> <li>• participantDefName--String</li> <li>• appSpecificBusObj--BusObj</li> <li>• genericBusObjList--BusObj, BusObjArray</li> </ul>
Maintain Simple Identity Relationship	<p>Maintains a simple identity relationship from within either a parent or child map.</p> <p>API: IdentityRelationship.maintainSimpleIdentityRelationship()</p>	<p>Inputs:</p> <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• relDefName--String</li> <li>• participantDefName--String</li> <li>• appSpecificBusObj--BusObj</li> <li>• genericBusObj--BusObj</li> </ul>



Table 37. General/APIs/Identity Relationship (continued)

Name	Description	Inputs and outputs with acceptable values
Update My Children	<p>Adds and deletes child instances in a specified parent/child relationship of an identity relationship, as necessary.</p> <p>API: IdentityRelationship.updateMyChildren()</p>	<p>Inputs:</p> <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• parentChildRelDefName--String</li> <li>• parentParticipantDef--String</li> <li>• parentBusObj--BusObj</li> <li>• childParticipantDef--String</li> <li>• childAttrName--String</li> <li>• childIdentityRelDefName--String</li> <li>• childIdentityParticipantDefName--String</li> </ul>

Table 38. General/APIs/Maps

Name	Description	Inputs and outputs with acceptable values
Get Adapter Name	<p>Retrieves the adapter name associated with the current map instance.</p> <p>API: MapExeContext.getConnName()</p>	<p>Inputs: map--BaseDLM</p> <p>Outputs: adapter name--String</p>
Get Calling Context	<p>Retrieves the calling context associated with the current map instance.</p> <p>API: MapExeContext.getInitiator()</p>	<p>Inputs: map--BaseDLM</p> <p>Outputs: calling context--String</p>
Get Original Request Business Object	<p>Retrieves the original-request business object associated with the current map instance.</p> <p>API: MapExeContext.getOriginalRequestBO()</p>	<p>Inputs: map--BaseDLM</p> <p>Outputs: original business object--BusObj</p>

Table 39. General/APIs/Maps/Constants

Name	Description	Inputs and outputs with acceptable values
Calling Context: ACCESS_REQUEST	<p>An access client has sent an access request from an external application to InterChange Server.</p> <p>API: MapExeContext.ACCESS_REQUEST</p>	Outputs: ACCESS_REQUEST--String
Calling Context: ACCESS_RESPONSE	<p>The source business object is sent back to the source access client in response to a subscription delivery request.</p> <p>API: MapExeContext.ACCESS_RESPONSE</p>	Outputs: ACCESS_RESPONSE--String
Calling Context: EVENT_DELIVERY	<p>A connector has sent an event from the application to InterChange Server (event-triggered flow).</p> <p>API: MapExeContext.EVENT_DELIVERY</p>	Outputs: EVENT_DELIVERY--String
Calling Context: SERVICE_CALL_FAILURE	<p>A collaboration's service call request has failed. As such, corrective action might need to be performed.</p> <p>API: MapExeContext.SERVICE_CALL_FAILURE</p>	Outputs: SERVICE_CALL_FAILURE--String

Table 39. General/APIs/Maps/Constants (continued)

Name	Description	Inputs and outputs with acceptable values
Calling Context: SERVICE_CALL_REQUEST	A collaboration is sending a business object down to the application through a service call request.  API: MapExeContext.SERVICE_CALL_REQUEST	Outputs: SERVICE_CALL_REQUEST --String
Calling Context: SERVICE_CALL_RESPONSE	A business object was received from the application as a result of a successful response to a collaboration service call request.  API: MapExeContext.SERVICE_CALL_RESPONSE	Outputs: SERVICE_CALL_RESPONSE --String

Table 40. General/APIs/Maps/Exception

Name	Description	Inputs and outputs with acceptable values
Raise Map Exception	Raises a map run-time exception.  API: raiseException()	Inputs: • map--BaseDLM • exception type--String • message--String
Raise Map Exception 1	Raises a map run-time exception.  API: raiseException()	Inputs: • map--BaseDLM • exception type--String • message--String • parameter 1--String
Raise Map Exception 2	Raises a map run-time exception.  API: raiseException()	Inputs: • map--BaseDLM • exception type--String • message--String • parameter 1--String • parameter 2--String
Raise Map Exception 3	Raises a map run-time exception.  API: raiseException()	Inputs: • map--BaseDLM • exception type--String • message--String • parameter 1--String • parameter 2--String • parameter 3--String
Raise Map Exception 4	Raises a map run-time exception.  API: raiseException()	Inputs: • map--BaseDLM • exception type--String • message--String • parameter 1--String • parameter 2--String • parameter 3--String • parameter 4--String

Table 40. General/APIs/Maps/Exception (continued)

Name	Description	Inputs and outputs with acceptable values
Raise Map Exception 5	Raises a map run-time exception. API: raiseException()	Inputs: <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• exception type--String</li> <li>• message--String</li> <li>• parameter 1--String</li> <li>• parameter 2--String</li> <li>• parameter 3--String</li> <li>• parameter 4--String</li> <li>• parameter 5--String</li> </ul>
Raise Map RunTimeEntity Exception	Raises a map run-time exception. API: raiseException()	Inputs: <ul style="list-style-type: none"> <li>• map--BaseDLM</li> <li>• exception--RunTimeEntityException</li> </ul>

Table 41. General/APIs/Participant

Name	Description	Inputs and outputs with acceptable values
Get Boolean Data	Retrieves the data associated with the Participant object. API: Participant.getBoolean()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--boolean
Get Business Object Data	Retrieves the data associated with the Participant object. API: Participant.getBusObj()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--BusObj
Get Double Data	Retrieves the data associated with the Participant object. API: Participant.getDouble()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--double
Get Float Data	Retrieves the data associated with the Participant object. API: Participant.getFloat()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--float
Get Instance ID	Retrieves the relationship instance ID of the relationship in which the participant instance is participating. API: Participant.getInstanceId()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: instance ID--int
Get Int Data	Retrieves the data associated with the Participant object. API: Participant.getInt()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--int
Get Long Data	Retrieves the data associated with the Participant object. API: Participant.getLong()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--long
Get Participant Name	Retrieves the participant definition name from which the participant instance is created. API: Participant.getParticipantDefinition()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: name--String

Table 41. General/APIs/Participant (continued)

Name	Description	Inputs and outputs with acceptable values
Get Relationship Name	Retrieves the name of the relationship definition in which the participant instance is participating.  API: Participant.getRelationshipDefinition()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: name--String
Get String Data	Retrieves the data associated with the Participant object.  API: Participant.getString()	Inputs: participant--Server.RelationshipServices.Participant  Outputs: data--String
New Participant	Creates a new participant instance with no relationship instance.  API: Participant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Output: participant--Server.RelationshipServices.Participant
New Participant in Relationship	Creates a new participant instance for adding to an existing participant in a relationship instance.  API: RelationshipServices.Participant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Output: participant--Server.RelationshipServices.Participant
Set Data	Sets the data associated with the participant instance.  API: Participant.set()	Inputs: <ul style="list-style-type: none"> <li>participant--Server.RelationshipServices.Participant</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul>
Set Instance ID	Sets the instance ID of the relationship in which the participant instance is participating.  API: Participant.setInstanceId()	Inputs: <ul style="list-style-type: none"> <li>participant--Server.RelationshipServices.Participant</li> <li>id--int</li> </ul>
Set Participant Definition	Sets the participant definition name from which the participant instance is created.  API: Participant.setParticipantDefinition()	Inputs: <ul style="list-style-type: none"> <li>participant--Server.RelationshipServices.Participant</li> <li>partDefName--String</li> </ul>
Set Relationship Definition	Sets the relationship definition in which the participant instance is participating.  API: Participant.setRelationshipDefinition()	Inputs: <ul style="list-style-type: none"> <li>participant--Server.RelationshipServices.Participant</li> <li>relDefName--String</li> </ul>

Table 42. General/APIs/Participant/Array

Name	Description	Inputs and outputs with acceptable values
Get Participant At	Retrieves the element at the specified index in the participant array.	Inputs: <ul style="list-style-type: none"> <li>array-- Server.RelationshipServices.Participant[]</li> <li>index--int</li> </ul> Outputs: participant-- Server.RelationshipServices.Participant
New Participant Array	Creates a new participant array with the specified size.	Inputs: size--int Outputs: array-- Server.RelationshipServices.Participant[]
Set Participant At	Sets the element at the specified index in the participant array.	Inputs: <ul style="list-style-type: none"> <li>array-- Server.RelationshipServices.Participant[]</li> <li>index--int</li> <li>participant-- Server.RelationshipServices.Participant</li> </ul>
Size	Retrieves the size of the participant array.	Inputs: array-- Server.RelationshipServices.Participant[] Outputs: size--int

Table 43. General/APIs/Participant/Constants

Name	Description	Inputs and outputs with acceptable values
Participant: INVALID _INSTANCE_ID	Participant constant indicating the participant ID is invalid. API: Participant.INVALID_INSTANCE_ID	Outputs: INVALID_INSTANCE_ID--int

Table 44. General/APIs/Relationship

Name	Description	Inputs and outputs with acceptable values
Add Participant	Adds an existing participant object to a relationship instance. API: Relationship.addParticipant()	Inputs: participant-- Server.RelationshipServices.Participant Outputs: result instance ID--int
Add Participant Data	Adds a new participant to an existing relationship instance. API: Relationship.addParticipant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Outputs: result instance ID--int
Add Participant Data to New Relationship	Adds a participant to a new relationship instance. API: Relationship.addParticipant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Outputs: result instance ID--int

Table 44. General/APIs/Relationship (continued)

Name	Description	Inputs and outputs with acceptable values
Create Relationship	Creates a new relationship instance. API: Relationship.create()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Outputs: instance ID--int
Create Relationship with Participant	Creates a new relationship instance. API: Relationship.create()	Inputs: participant--Server.RelationshipServices.Participant Outputs: instance ID--int
Deactivate Participant	Deactivates a participant from one or more relationship instances. API: Relationship.deactivate Participant()	Inputs: participant--Server.RelationshipServices.Participant
Deactivate Participant By Data	Deactivates a participant from one or more relationship instances. API: Relationship.deactivate Participant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul>
Deactivate Participant By Instance	Deactivates a participant from a specific relationship instance. API: Relationship.deactivate ParticipantByInstance()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> </ul>
Deactivate Participant By Instance Data	Deactivates a participant from a specific relationship instance with the data associated with the participant. API: Relationship.deactivate ParticipantByInstance()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul>
Delete Participant	Removes a participant instance from one or more relationship instances. API: Relationship.deleteParticipant()	Inputs: participant--Server.RelationshipServices.Participant
Delete Participant By Instance	Removes a participant from a specific relationship instance. API: Relationship.deleteParticipanByInstancet()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> </ul>
Delete Participant By Instance Data	Removes a participant from a specific relationship instance with the data associated with the participant. API: Relationship.deleteParticipanByInstancet()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul>

Table 44. General/APIs/Relationship (continued)

Name	Description	Inputs and outputs with acceptable values
Delete Participant with Data	Removes a participant instance from one or more relationship instances.  API: Relationship.deleteParticipant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul>
Get Next Instance ID	Returns the next available relationship instance ID for a relationship, based on the relationship definition name.  API: Relationship.getNewID()	Inputs: relDefName--String  Outputs: ID--int
Retrieve Instances	Retrieves zero or more IDs of relationship instances which contain the given participant(s).  API: Relationship.retrieveInstances()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String,String[]</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Outputs: instance IDs--int
Retrieve Instances for Participant	Retrieves zero or more IDs of relationship instances which contain a given participant.  API: Relationship.retrieveInstances()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partData--BusObj, String, long, int, double, float, boolean</li> </ul> Outputs: instance IDs--int
Retrieve Participants	Retrieves zero or more participants from a relationship instance.  API: Relationship.retrieveParticipants()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String, String[]</li> <li>instanceId--int</li> </ul> Outputs: participant instances--Server.RelationshipServices.Participant[]
Retrieve Participants with ID	Retrieves zero or more participants from a relationship instance.  API: Relationship.retrieveParticipants()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>instanceId--int</li> </ul> Outputs: participant instances--Server.RelationshipServices.Participant[]
Update Participant	Updates a participant in one or more relationship instances.  API: Relationship.updateParticipant()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>partData--BusObj</li> </ul>
Update Participant By Instance	Updates a participant in a specific relationship instance.  API: Relationship.updateParticipantByInstance()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String, String[]</li> <li>instanceId--int</li> </ul>
Update Participant By Instance Data	Updates a participant in a specific relationship instance with the data associated with the participant.  API: Relationship.updateParticipantByInstance()	Inputs: <ul style="list-style-type: none"> <li>relDefName--String</li> <li>partDefName--String</li> <li>instanceId--int</li> <li>partData--BusObj, String</li> </ul>

Table 45. General/Date

Name	Description	Inputs and outputs with acceptable values
Add Day	Adds additional days to the from date.	Inputs: <ul style="list-style-type: none"> <li>• from date--String</li> <li>• date format--String</li> <li>• day to add--int</li> </ul> Outputs: to date-- String
Add Month	Adds additional months to the from date.	Inputs: <ul style="list-style-type: none"> <li>• from date--String</li> <li>• date format--String</li> <li>• month to add--int</li> </ul> Outputs: to date-- String
Add Year	Adds additional years to the from date.	Inputs: <ul style="list-style-type: none"> <li>• from date--String</li> <li>• date format--String</li> <li>• year to add--int</li> </ul> Outputs: to date-- String
Date After	Compares two dates and determines whether Date 1 is after Date 2.	Inputs: <ul style="list-style-type: none"> <li>• Date 1--String</li> <li>• Date 1 format--String</li> <li>• Date 2--String</li> <li>• Date 2 format--String</li> </ul> Outputs: Is Date 1 after Date 2?-- boolean
Date Before	Compares two dates and determines whether Date 1 is before Date 2.	Inputs: <ul style="list-style-type: none"> <li>• Date 1--String</li> <li>• Date 1 format--String</li> <li>• Date 2--String</li> <li>• Date 2 format--String</li> </ul> Outputs: Is Date 1 before Date 2?-- boolean
Date Equals	Compares two dates and determines whether they are equal.	Inputs: <ul style="list-style-type: none"> <li>• Date 1--String</li> <li>• Date 1 format--String</li> <li>• Date 2--String</li> <li>• Date 2 format--String</li> </ul> Outputs: Are they equal?-- boolean
Format Change	Changes a date format.	Inputs: <ul style="list-style-type: none"> <li>• date--String</li> <li>• input format--String</li> <li>• output format--String</li> </ul> Outputs: formatted date--String



Table 45. General/Date (continued)

Name	Description	Inputs and outputs with acceptable values
Get Day	Returns the numeric day of month based on date expression.	Inputs: <ul style="list-style-type: none"> <li>• Date--String</li> <li>• Format--String</li> </ul> Outputs: Day--int
Get Month	Returns the numeric month of year based on date expression.	Inputs: <ul style="list-style-type: none"> <li>• Date--String</li> <li>• Format--String</li> </ul> Outputs: Month--int
Get Year	Returns the numeric year based on date expression.	Inputs: <ul style="list-style-type: none"> <li>• Date--String</li> <li>• Format--String</li> </ul> Outputs: Year--int
Get Year Month Day	Given an input date, extracts the Year/Month/Day parts from the input date respectively.	Inputs: <ul style="list-style-type: none"> <li>• Date--String</li> <li>• Format--String</li> </ul> Outputs: <ul style="list-style-type: none"> <li>• Year--int</li> <li>• Month--int</li> <li>• Day--int</li> </ul>
Now	Gets today's date.	Inputs: format--String Outputs: now--String

Table 46. General/Date/Formats

Name	Description	Inputs and outputs with acceptable values
yyyy-MM-dd	Date format of yyyy-MM-dd <b>Example:</b> 2003-02-25	Outputs: format--String
yyyyMMdd	Date format of yyyyMMdd <b>Example:</b> 20030225	Outputs: format--String
yyyyMMdd HH:mm:ss	Date format of yyyyMMdd HH:mm:ss <b>Example:</b> 20030225 12:36:40	Outputs: format--String

Table 47. General/Logging and Tracing

Name	Description	Inputs and outputs with acceptable values
Log error	Sends the specified error message to the ICS log file.	Inputs: message--String, byte, short, int, long, float, double
Log error ID	Sends the error message associated with the specified ID to the ICS log file.	Inputs: ID--String, byte, short, int, long, float, double
Log information	Sends the specified information message to the ICS log file.	Inputs: message--String, byte, short, int, long, float, double
Log information ID	Sends the information message associated with the specified ID to the ICS log file.	Inputs: ID--String, byte, short, int, long, float, double

Table 47. General/Logging and Tracing (continued)

Name	Description	Inputs and outputs with acceptable values
Log warning	Sends the specified warning message to the ICS log file	Inputs: message--String, byte, short, int, long, float, double
Log warning ID	Sends the warning message associated with the specified ID to the ICS log file.	Inputs: ID--String, byte, short, int, long, float, double
Trace	Sends the specified trace message to the ICS log file.	Inputs: message--String, byte, short, int, long, float, double

Table 48. General/Logging and Tracing/Log Error

Name	Description	Inputs and outputs with acceptable values
Log error ID 1	Formats the error message associated with the specified ID with the parameter and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>parameter--String, byte, short, int, long, float, double</li> </ul>
Log error ID 2	Formats the error message associated with the specified ID with the parameters and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>parameter 1--String, byte, short, int, long, float, double</li> <li>parameter 2--String, byte, short, int, long, float, double</li> </ul>
Log error ID 3	Formats the error message associated with the specified ID with the parameters and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>parameter 1--String, byte, short, int, long, float, double</li> <li>parameter 2--String, byte, short, int, long, float, double</li> <li>parameter 3--String, byte, short, int, long, float, double</li> </ul>

Table 49. General/Logging and Tracing/Log Information

Name	Description	Inputs and outputs with acceptable values
Log information ID 1	Formats the information message associated with the specified ID with the parameter and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>parameter--String, byte, short, int, long, float, double</li> </ul>
Log information ID 2	Formats the information message associated with the specified ID with the parameters and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>parameter 1--String, byte, short, int, long, float, double</li> <li>parameter 2--String, byte, short, int, long, float, double</li> </ul>

Table 49. General/Logging and Tracing/Log Information (continued)

Name	Description	Inputs and outputs with acceptable values
Log information ID 3	Formats the information message associated with the specified ID with the parameters and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>• ID--String, byte, short, int, long, float, double</li> <li>• parameter 1--String, byte, short, int, long, float, double</li> <li>• parameter 2--String, byte, short, int, long, float, double</li> <li>• parameter 3--String, byte, short, int, long, float, double</li> </ul>

Table 50. General/Logging and Tracing/Log Warning

Name	Description	Inputs and outputs with acceptable values
Log warning ID 1	Formats the warning message associated with the specified ID with the parameter and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>• ID--String, byte, short, int, long, float, double</li> <li>• parameter--String, byte, short, int, long, float, double</li> </ul>
Log warning ID 2	Formats the warning message associated with the specified ID with the parameters and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>• ID--String, byte, short, int, long, float, double</li> <li>• parameter 1--String, byte, short, int, long, float, double</li> <li>• parameter 2--String, byte, short, int, long, float, double</li> </ul>
Log warning ID 3	Formats the warning message associated with the specified ID with the parameters and sends it to the ICS log file.	Inputs: <ul style="list-style-type: none"> <li>• ID--String, byte, short, int, long, float, double</li> <li>• parameter 1--String, byte, short, int, long, float, double</li> <li>• parameter 2--String, byte, short, int, long, float, double</li> <li>• parameter 3--String, byte, short, int, long, float, double</li> </ul>

Table 51. General/Logging and Tracing/Trace

Name	Description	Inputs and outputs with acceptable values
Trace ID 1	Formats the trace message associated with the specified ID with the parameter and displays it if tracing is set to the specified level or a higher level.	Inputs: <ul style="list-style-type: none"> <li>• ID--String, byte, short, int, long, float, double</li> <li>• level--String, byte, short, int, long, float, double</li> <li>• parameter--String, byte, short, int, long, float, double</li> </ul>

Table 51. General/Logging and Tracing/Trace (continued)

Name	Description	Inputs and outputs with acceptable values
Trace ID 2	Formats the trace message associated with the specified ID with the parameters and displays it if tracing is set to the specified level or a higher level.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>level--String, byte, short, int, long, float, double</li> <li>parameter 1--String, byte, short, int, long, float, double</li> <li>parameter 2--String, byte, short, int, long, float, double</li> </ul>
Trace ID 3	Formats the trace message associated with the specified ID with the parameters and displays it if tracing is set to the specified level or a higher level.	Inputs: <ul style="list-style-type: none"> <li>ID--String, byte, short, int, long, float, double</li> <li>level--String, byte, short, int, long, float, double</li> <li>parameter 1--String, byte, short, int, long, float, double</li> <li>parameter 2--String, byte, short, int, long, float, double</li> <li>parameter 3--String, byte, short, int, long, float, double</li> </ul>
Trace on Level	Displays the trace message if tracing is set to the specified level or a higher level.	Inputs: <ul style="list-style-type: none"> <li>message--String, byte, short, int, long, float, double</li> <li>level--String, byte, short, int, long, float, double</li> </ul>

Table 52. General/Mapping

Name	Description	Inputs and outputs with acceptable values
Run Map	Executes the specified map with the current calling context.	Inputs: <ul style="list-style-type: none"> <li>Map Name--String</li> <li>Source Business Objects--BusObj, BusObj[]</li> </ul> Outputs: Map Results--BusObj, BusObj[]
Run Map with Context	Executes the specified map with the calling context specified.	Inputs: <ul style="list-style-type: none"> <li>Map Name--String</li> <li>Source Business Objects--BusObj, BusObj[]</li> <li>calling context--String</li> </ul> Outputs: Map Results--BusObj, BusObj[]

Table 53. General/Math

Name	Description	Inputs and outputs with acceptable values
Absolute value	a=abs(b) API: Math.abs()	Inputs: b--byte, short, int, long, float, double Outputs: a--byte, short, int, long, float, double
Ceiling	Returns the next highest integer that is greater than or equal to the specified numeric expression.	Inputs: number--String, float, double Outputs: ceiling--int

Table 53. General/Math (continued)

Name	Description	Inputs and outputs with acceptable values
Divide	$a=b/c$	Inputs: <ul style="list-style-type: none"> <li>• b--byte, short, int, long, float, double</li> <li>• c--byte, short, int, long, float, double</li> </ul> Outputs: a--byte, short, int, long, float, double
Equal	Is value 1 equal to value 2?	Inputs: <ul style="list-style-type: none"> <li>• value 1--String, byte, short, int, long, float, double</li> <li>• value 2--String, byte, short, int, long, float, double</li> </ul> Outputs: are they equal?--boolean
Floor	Returns the next lowest integer that is greater than or equal to the specified numeric expression.	Inputs: number--String, float, double Outputs: floor--int
Greater than	Is value 1 greater than value 2?	Inputs: <ul style="list-style-type: none"> <li>• value 1--byte, short, int, long, float, double</li> <li>• value 2--byte, short, int, long, float, double</li> </ul> Outputs: result--boolean
Greater than or Equal	Is value 1 greater than or equal to value 2?	Inputs: <ul style="list-style-type: none"> <li>• value 1--byte, short, int, long, float, double</li> <li>• value 2--byte, short, int, long, float, double</li> </ul> Outputs: result--boolean
Less than	result=value 1 is less than value 2?	Inputs: <ul style="list-style-type: none"> <li>• value 1--byte, short, int, long, float, double</li> <li>• value 2--byte, short, int, long, float, double</li> </ul> Outputs: result--boolean
Less than or equal	Is value 1 less than or equal to value 2?	Inputs: <ul style="list-style-type: none"> <li>• value 1--byte, short, int, long, float, double</li> <li>• value 2--byte, short, int, long, float, double</li> </ul> Outputs: result--boolean
Maximum	$a=\max(b, c)$ API: Math.max()	Inputs: <ul style="list-style-type: none"> <li>• b--byte, short, int, long, float, double</li> <li>• c--byte, short, int, long, float, double</li> </ul> Outputs: a--byte, short, int, long, float, double
Minimum	$a=\min(b, c)$ API: Math.min()	Inputs: <ul style="list-style-type: none"> <li>• b--byte, short, int, long, float, double</li> <li>• c--byte, short, int, long, float, double</li> </ul> Outputs: a--byte, short, int, long, float, double

Table 53. General/Math (continued)

Name	Description	Inputs and outputs with acceptable values
Minus	$a=b-c$	Inputs: <ul style="list-style-type: none"> <li>• b--byte, short, int, long, float, double</li> <li>• c--byte, short, int, long, float, double</li> </ul> Outputs: a--byte, short, int, long, float, double
Multiply	$a=b*c$	Inputs: <ul style="list-style-type: none"> <li>• b--byte, short, int, long, float, double</li> <li>• c--byte, short, int, long, float, double</li> </ul> Outputs: a--byte, short, int, long, float, double
Not Equal	result=is value 1 not equal to value 2?	Inputs: <ul style="list-style-type: none"> <li>• value 1--String, byte, short, int, long, float, double</li> <li>• value 2--String, byte, short, int, long, float, double</li> </ul> Outputs: are they not equal?--boolean
Not a Number	Returns true if input is not a number.	Inputs: input--String Outputs: is not a number--boolean
Number to String	Converts a numeric expression to a character expression.	Inputs: number--String, short, int, long, float, double Outputs: string--String
Plus	$a=b+c$	Inputs: <ul style="list-style-type: none"> <li>• b--byte, short, int, long, float, double</li> <li>• c--byte, short, int, long, float, double</li> </ul> Outputs: a--byte, short, int, long, float, double
Round	Rounds a numeric expression down to the next lowest integer if <5; otherwise, the integer is rounded up.	Inputs: number--String, float, double Outputs: rounded number--int
String to Number	Converts a character expression to a numeric expression.  API: Math.type()	Inputs: string--String Outputs: String, short, int, long, float, double

Table 54. General/Properties

Name	Description	Inputs and outputs with acceptable values
Get Property	Retrieves the specified configuration property value.	Inputs: property name--String Outputs: property value--String

Table 55. General/Relationship

Name	Description	Inputs and outputs with acceptable values
Maintain Identity Relationship	Maintains an identity relationship with the maintainSimpleIdentityRelationship() Relationship API.	Inputs: <ul style="list-style-type: none"> <li>relationship name--String</li> <li>participant name--String</li> <li>Generic Business Object--String</li> <li>Application-Specific Business Object--String</li> <li>calling context--String</li> </ul>
Static Lookup	Looks up a static value in the relationship.	Inputs: <ul style="list-style-type: none"> <li>relationship name--String</li> <li>participant name--String</li> <li>inbound?--boolean</li> <li>source value--String</li> </ul> Outputs: lookup value--String

Table 56. General/String

Name	Description	Inputs and outputs with acceptable values
Append Text	Appends the "in string2" to the end of the string "in string 1."	Inputs: <ul style="list-style-type: none"> <li>in string 1--String</li> <li>in string 2--String</li> </ul> Outputs: result--String
If	Returns the first value if condition is true and the second value if condition is false.	Inputs: <ul style="list-style-type: none"> <li>condition--boolean, Boolean</li> <li>value 1--String</li> <li>value 2--String</li> </ul> Outputs: result--String
Is Empty	Returns the second value if the first value is empty.	Inputs: <ul style="list-style-type: none"> <li>value 1--String</li> <li>value 2--String</li> </ul> Outputs: result--String
Is NULL	Returns the second value if the first value is null.	Inputs: <ul style="list-style-type: none"> <li>value 1--String</li> <li>value 2--String</li> </ul> Outputs: result--String
Left Fill	Returns a string of the specified length; fills the left with indicated value.	Inputs: <ul style="list-style-type: none"> <li>string--String</li> <li>fill string--String</li> <li>length--int</li> </ul> Outputs: filled string--String

Table 56. General/String (continued)

Name	Description	Inputs and outputs with acceptable values
Left String	Returns the left portion of string for the specified number of positions.	Inputs: <ul style="list-style-type: none"> <li>string--String</li> <li>length--int</li> </ul> Outputs: left string--String
Lower Case	Changes all characters to Lower Case letters	Inputs: fromString--String Outputs: toString--String
Object To String	Gets a string representation of the object.	Inputs: object--Object Outputs: string--String
Repeat	Returns a character string that contains a specified character expression repeated a specified number of times.	Inputs: <ul style="list-style-type: none"> <li>repeating string--String</li> <li>repeat count--int</li> </ul> Outputs: result--String
Replace	Replaces part of a string with indicated value data.	Inputs: <ul style="list-style-type: none"> <li>input--String</li> <li>old string--String</li> <li>new string--String</li> </ul> Outputs: replaced string--String
Right Fill	Returns a string of the specified length; fills the right with indicated value.	Inputs: <ul style="list-style-type: none"> <li>string--String</li> <li>fill string--String</li> <li>length--int</li> </ul> Outputs: filled string--String
Right String	Returns the right portion of string for the specified number of positions.	Inputs: <ul style="list-style-type: none"> <li>string--String</li> <li>length--int</li> </ul> Outputs: right string--String
Substring by position	Returns a portion of the string based on start and end parameters.	Inputs: <ul style="list-style-type: none"> <li>string--String</li> <li>start position--int</li> <li>end position--int</li> </ul> Outputs: substring--String
Substring by value	Returns a portion of the string based on start and end parameters. The substring will not include the start and end value.	Inputs: <ul style="list-style-type: none"> <li>string--String</li> <li>start value--int</li> <li>end value--int</li> </ul> Outputs: substring--String
Text Equal	Compares the strings "inString1" and "inString2" and determine whether they are the same.	Inputs: <ul style="list-style-type: none"> <li>inString1--String</li> <li>inString2--String</li> </ul> Outputs: are they equal?--boolean



Table 56. General/String (continued)

Name	Description	Inputs and outputs with acceptable values
Text Equal Ignore Case	Compares the strings "inString1" and "inString2" lexicographically, ignoring case considerations.	Inputs: <ul style="list-style-type: none"> <li>inString1--String</li> <li>inString2--String</li> </ul> Outputs: are they equal?--boolean
Text Length	Finds the total number of characters in a string	Inputs: str--String Outputs: length--byte, short, int, long, float, double
Trim Left	Trims the specified number of characters from the left side of the string.	Inputs: <ul style="list-style-type: none"> <li>input--String</li> <li>trim length--int</li> </ul> Outputs: result--String
Trim Right	Trims the specified number of characters from the right side of the string.	Inputs: <ul style="list-style-type: none"> <li>input--String</li> <li>trim length--int</li> </ul> Outputs: result--String
Trim Text	Trims white spaces before and after the string	Inputs: in string--String Outputs: trimmed string--String
Upper Case	Changes all characters into Upper Case letters	Inputs: fromString--String Outputs: toString--String

Table 57. General/Utilities

Name	Description	Inputs and outputs with acceptable values
Catch Error	Catches all the Exceptions thrown in the current activity and its subactivities. (Double-click the function block icon in the canvas to define the subactivity.)	Inputs: <ul style="list-style-type: none"> <li>Error Name--String</li> <li>Error Message--String</li> </ul>
Catch Error Type	Catches the specified Exception type thrown in the current activity and its subactivities. (Double-click the function block icon in the canvas to define the subactivity.)	Inputs: <ul style="list-style-type: none"> <li>error type--String</li> <li>Error Message--String</li> </ul>
Condition	If "Condition" is true, executes the subactivity defined in "True Action"; otherwise, executes the subactivity defined in "False Action." (Double-click the function block icon in the canvas to define the subactivity.)	Inputs: Condition--boolean
Loop	Repeats the subactivity until "Condition" is false. (Double-click the function block icon in the canvas to define the subactivity.)	Inputs: Condition--boolean

Table 57. General/Utilities (continued)

Name	Description	Inputs and outputs with acceptable values
Move Attribute in Child	Moves the value from "from attribute" to "to attribute".	Inputs: <ul style="list-style-type: none"> <li>source parent--BusObj</li> <li>source child BO attribute--string</li> <li>from attribute--String</li> <li>destination parent--BusObj</li> <li>destination child BO attribute--String</li> <li>to attribute--String</li> </ul>
Raise Error	Throws a new Java Exception with the given message.	Inputs: message--String
Raise Error Type	Throws the specified Java Exception with the given message.	Inputs: <ul style="list-style-type: none"> <li>error type--String</li> <li>message--String</li> </ul>

Table 58. General/Utilities/Vector

Name	Description	Inputs and outputs with acceptable values
Add Element	Adds the specified element to the end of the vector, increasing its size by one.	Inputs: vector--java.util.Vector Outputs: element--Object
Get Element	Gets the element at the specified index in the Vector object.	Inputs: <ul style="list-style-type: none"> <li>vector--java.util.Vector</li> <li>index--int</li> </ul> Outputs: element--Object
Iterate Vector	Iterates through the vector object.	Inputs: <ul style="list-style-type: none"> <li>vector--java.util.Vector</li> <li>current index--int</li> <li>current element--Object</li> </ul>
New Vector	Creates a new vector object.	Outputs: vector--java.util.Vector
Size	Gets the number of elements in this vector.	Inputs: vector--java.util.Vector Outputs: size--int
To Array	Gets the array representation containing all of the elements in this vector.	Inputs: vector--java.util.Vector Outputs: array--Object[]

## Example 1: Steps for changing a value to uppercase

The following example illustrates the steps for using Activity Editor to change the source attribute's value to all uppercase and assign the change to the destination attribute.

Perform the following steps:

1. From the Diagram tab, drag the source attribute onto the destination attribute to create a Custom transformation rule. Then click the icon of the Custom transformation rule to open Activity Editor.

**Result:** Activity Editor opens.

**Example:** Figure 58 shows the Custom transformation we are using in this example. The source attribute is ObjClarify\_contact.LastName, and the

destination attribute is ObjContact.LastName.

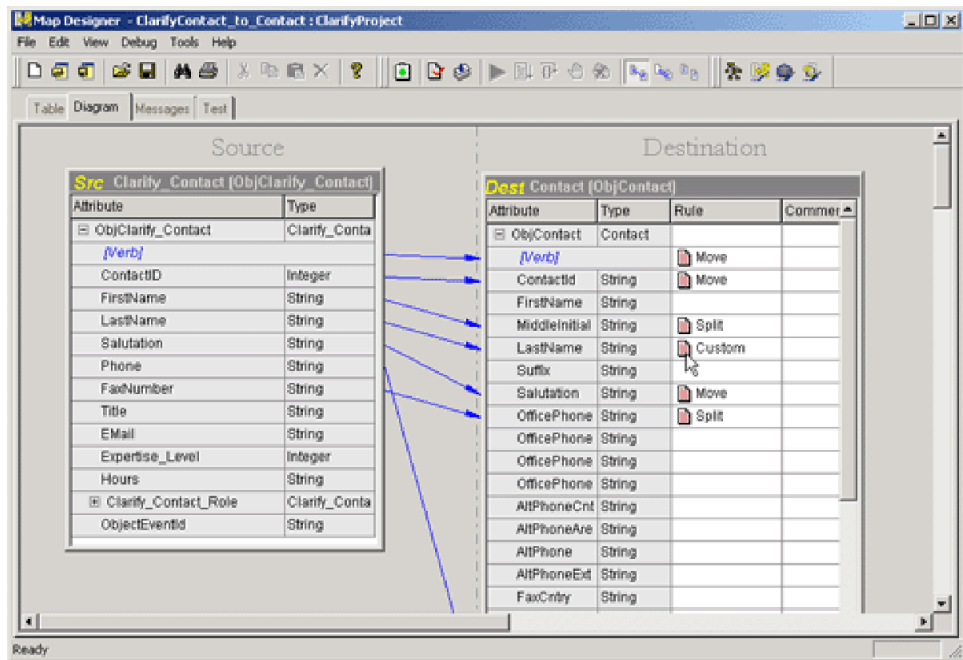


Figure 58. Custom transformation rule

For more information on creating Custom and other transformations, see Chapter 2, "Creating maps," on page 15.

2. Select a category in the Library window (top left) to show the function blocks available in that category, as icons, in the Content window (bottom left).

Figure 59 shows the available function blocks for the "String" category; the source and destination attributes in this example are displayed as icons in the editing canvas.

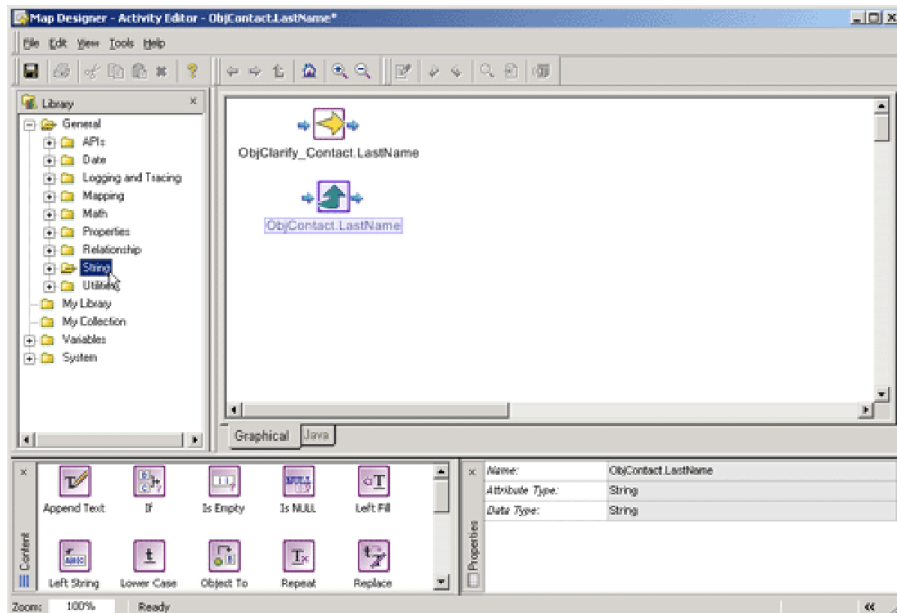


Figure 59. Function blocks in String category and icons for source and destination attributes

3. To use any of the function blocks in the activity, drag the function block from the tree in the Library window and drop it onto the editing canvas; or alternatively, drag the icon from the Content window and drop it onto the editing canvas.

**Example:** In our example, we want to change the source attribute to all uppercase letters, so we will drag-and-drop the Upper Case function block in the String category from the Content window onto the editing canvas. This action is shown in Figure 60.

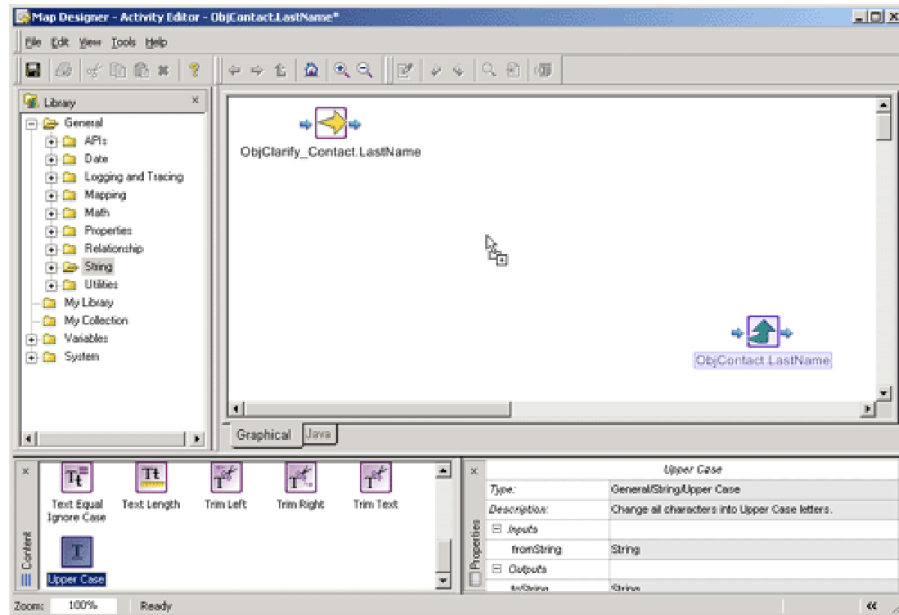


Figure 60. Dragging the Upper Case function block

4. After you drop a function block on the editing canvas, you can move it around the canvas by selecting the function block icon and dragging and dropping it at the desired position. When the function block is in place, you are ready to connect the inputs and outputs of the function block to define the flow of execution.

**Example:** In our example, we want to convert the attribute value of `ObjClarify_Contact.LastName` to all uppercase letters. We can do this by connecting the output of the icon for `ObjClarify_Contact.LastName` to the input of the Upper Case function block. To do this, move the mouse cursor to the output of the icon of port `ObjClarify_Contact.LastName`.

**Result:** The shape of the icon will change to an arrow to indicate that you can initiate a link at that point, as shown in Figure 61.

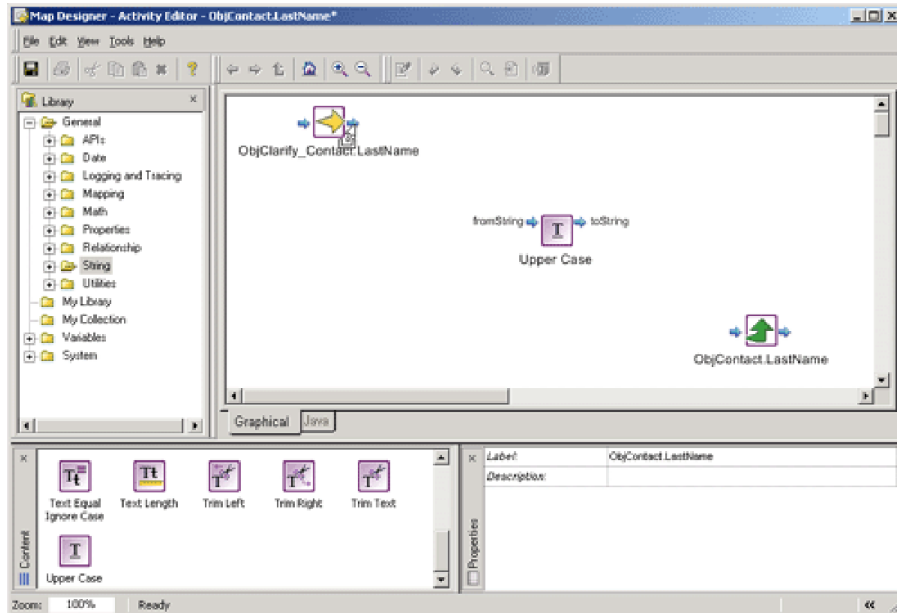


Figure 61. Cursor as arrow at output port of *ObjClarify\_Contact.LastName*

5. When the mouse icon is changed to an arrow, hold down the mouse button and move the mouse to the input of the Upper Case function block, and release the mouse button. A connection link will be drawn to connect the input and outputs.

To indicate that the result of the Upper Case function block should be assigned to the destination attribute (in our example, *ObjContact.LastName*), repeat the same steps to drag-and-drop from the output of the Upper Case function block to the input of the *ObjContact.LastName* port icon. Figure 62 shows the connection links.

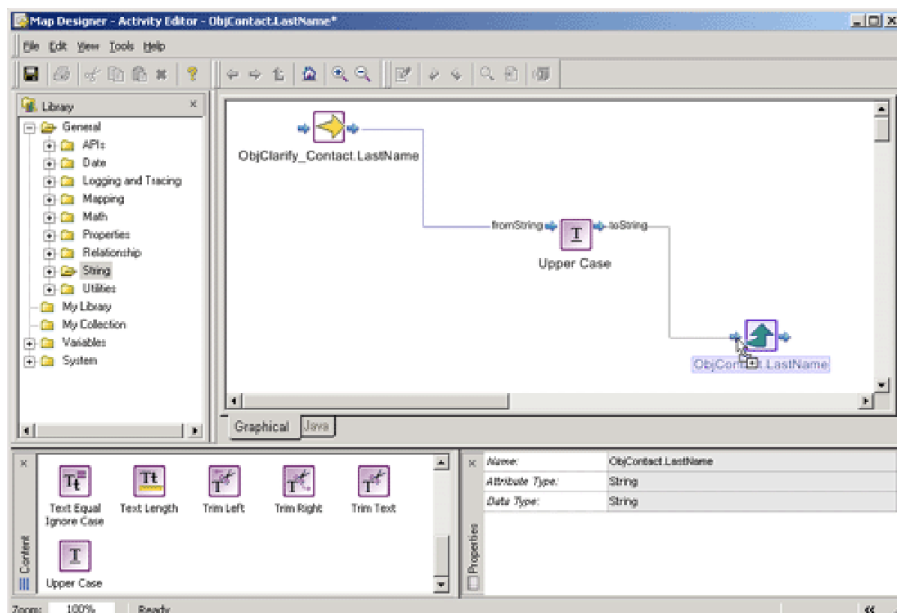
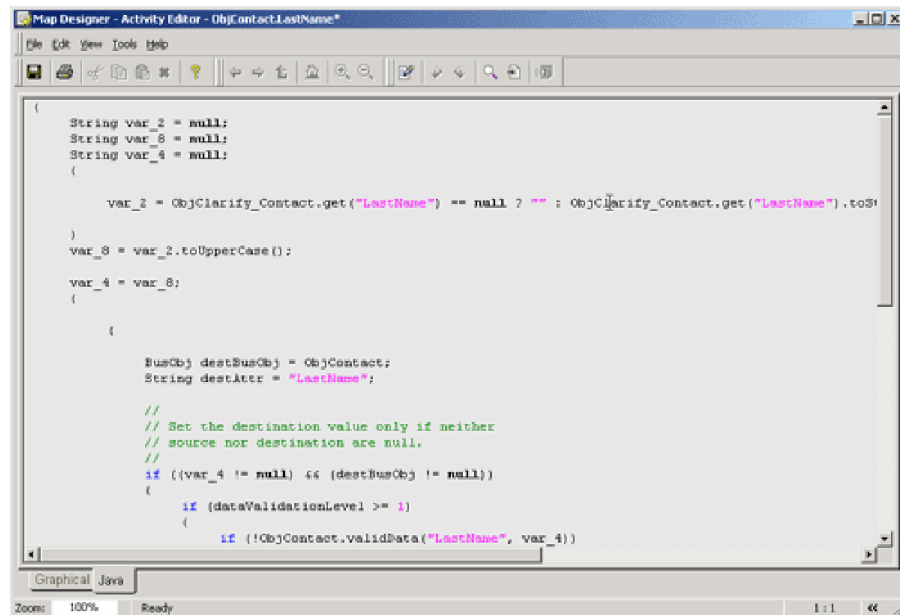


Figure 62. Upper Case function block with connection links

**Result:** We have defined an activity which will take the value of the source attribute, uppercase it, and set the uppercased value to the destination attribute.

6. Save the activity by selecting To Project or To File from the File > Save submenu or by clicking the Save Map to Project or Save Map to File button in the Standard toolbar.
7. To see an example of the Java code that will be generated by this activity, click the Java tab.

**Result:** The Java tab will be activated with the sample Java code, as shown in Figure 63.



```
(
String var_2 = null;
String var_8 = null;
String var_4 = null;
{
    var_2 = ObjClarify_Contact.get("LastName") == null ? "" : ObjClarify_Contact.get("LastName").toUpperCase();
}
var_8 = var_2.toUpperCase();
var_4 = var_8;
{
    BusObj destBusObj = ObjContact;
    String destAttr = "LastName";

    // Set the destination value only if neither
    // source nor destination are null.
    //
    if ((var_4 != null) && (destBusObj != null))
    {
        if (dataValidationLevel >= 1)
        {
            if (!ObjContact.isValidData("LastName", var_4))

```

Figure 63. Java tab with code

## Example 2: Steps for changing a date format

The following example illustrates the steps for using Activity Editor to change the source value's date format to a different format and assign it to the destination attribute.

Perform the following steps:

1. From the Diagram tab, drag the source attribute onto the destination attribute to create a Custom transformation rule. Then click the icon of the Custom transformation rule to open Activity Editor.

**Result:** Activity Editor opens.

**Example:** Figure 64 shows the Custom transformation we are using in this example. The source attribute is ObjClarify\_QuoteSchedule.PriceProgExpireDate, and the destination attribute

is ObjARInvoice.GLPostingDate.

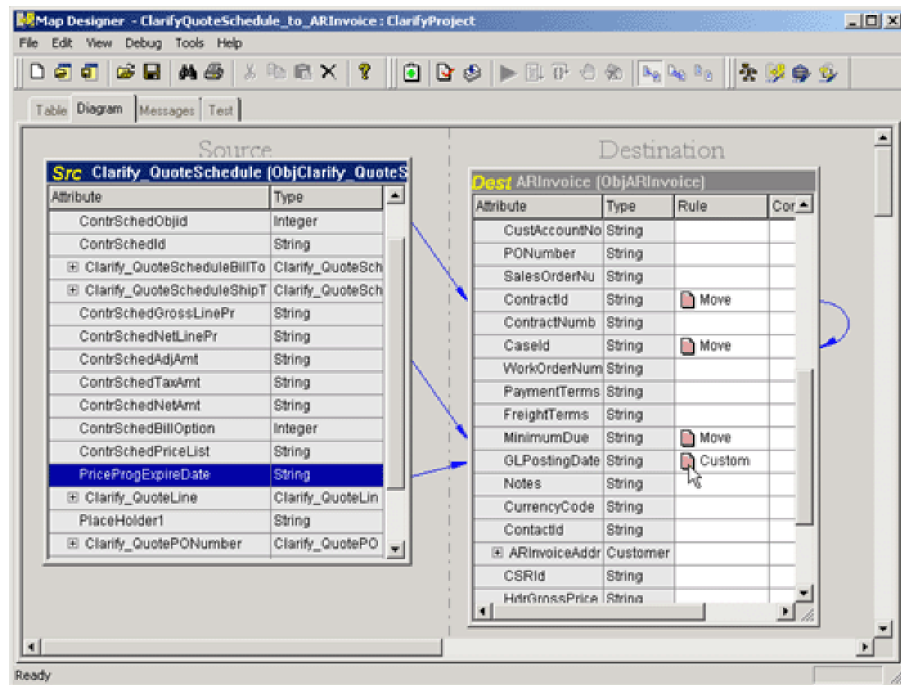


Figure 64. Custom transformation rule

For more information on creating Custom and other transformations, see Chapter 2, “Creating maps,” on page 15.

2. Select a category in the Library window (top left) to show the available function blocks available in that category, as icons, in the Content window (bottom left).

Figure 65 shows the available functions blocks for the “Date” category; the source and destination attributes in our example are displayed as icons in the

editing canvas.

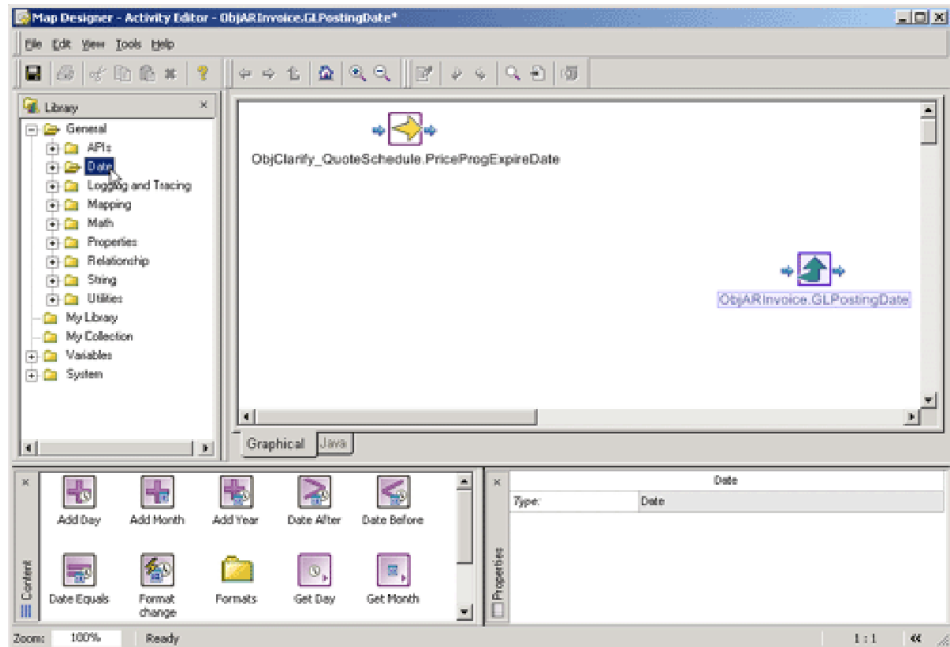


Figure 65. Function blocks in Date category and icons for source and destination attributes

3. To use any of the function blocks in the activity, drag the function block from the tree in the Library window and drop it onto the editing canvas; or alternatively, drag the icon from the Content window and drop it onto the activity canvas.

**Example:** We want to change the date format of the source attribute from "yyyyMMdd" to "yyyy.MM.dd G 'at' HH:mm:ss z" and assign it to the destination attribute; so we will drag-and-drop the Format Change function block in the Date category from the Content window onto the editing canvas, as shown in Figure 66.

**Note:** A date formatted with "yyyyMMdd" looks like this: "20030227"; a date formatted with "yyyy.MM.dd G 'at' HH:mm:ss z" looks like this



"2003.02.27 AD at 00:00:00 PDT".

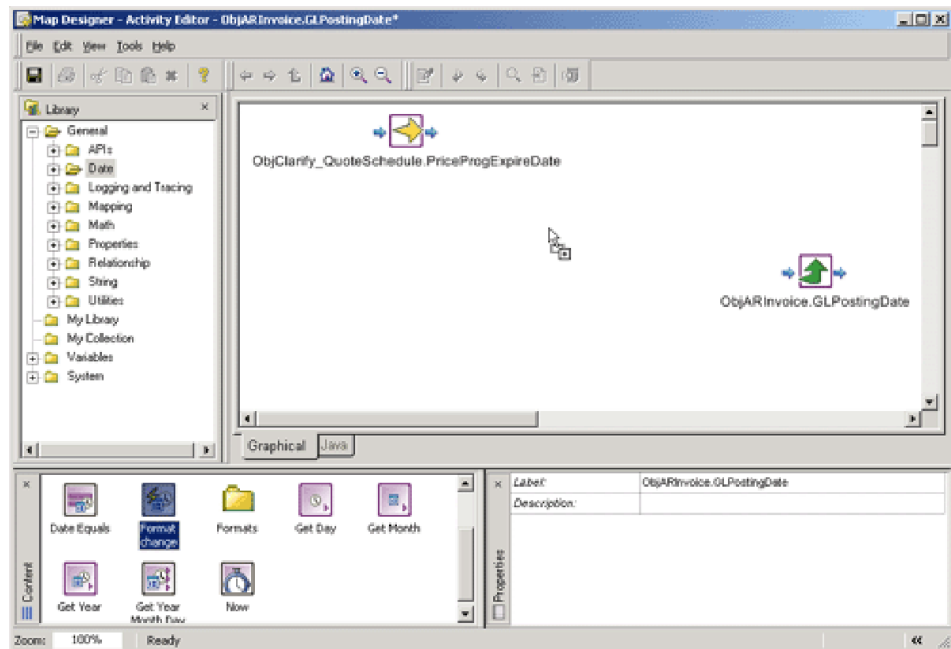


Figure 66. Dragging the Date Format Change function block

4. After you drop a function block onto the activity canvas, you can move it around the canvas by selecting the function block icon and dragging and dropping it at the desired position. When the function block is in place, you are ready to connect the inputs and outputs of the function block to define the flow of execution.

**Example:** We want to change the date format of the source attribute `ObjClarify_QuoteSchedule.PriceProgExpireDate`. We will do this by connecting the output of the port icon for `ObjClarify_QuoteSchedule.PriceProgExpireDate` to the date input of the `Format Change` function block. To do this, move the mouse cursor to the output of the icon of port `ObjClarify_QuoteSchedule.PriceProgExpireDate`.

**Result:** The shape of the icon will change to an arrow to indicate that you can initiate a link at that point, as shown in Figure 67.

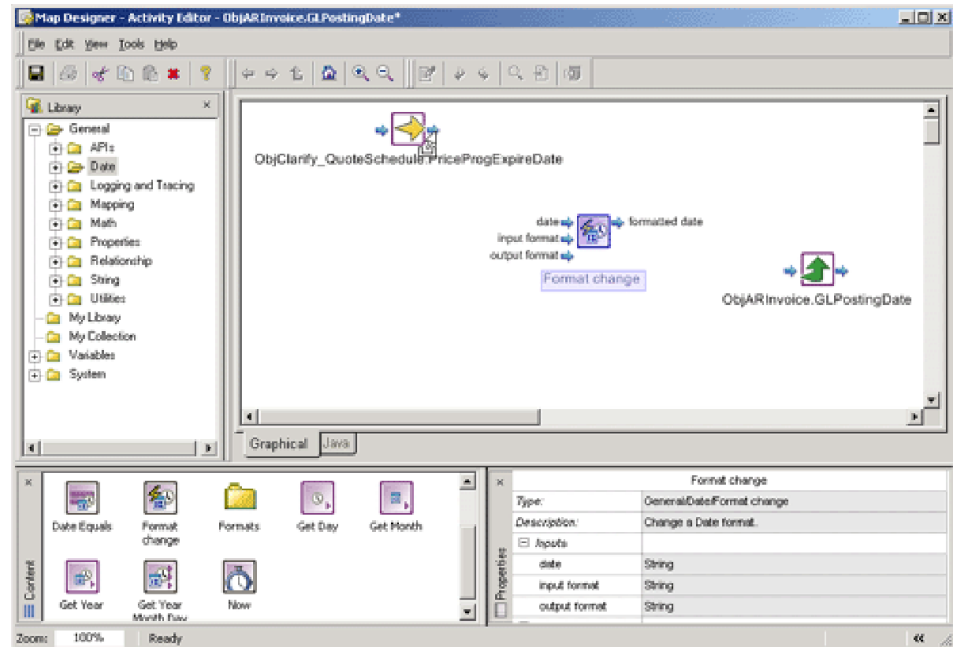


Figure 67. Cursor as arrow at output port of ObjClarify\_QuoteSchedule.PriceProgExpireDate

5. When the mouse icon is changed to an arrow, hold down the mouse button and move the mouse to the date input of the Format Change function block, and release the mouse button. A connection link will be drawn to connect the input and outputs.

To indicate that the result of the Format Change function block should be assigned to the destination attribute ObjARInvoice.GLPostingDate, repeat the same steps to drag-and-drop from the output of the Format Change function block to the input of the ObjARInvoice.GLPostingDate port icon. Figure 68 shows the connection links.

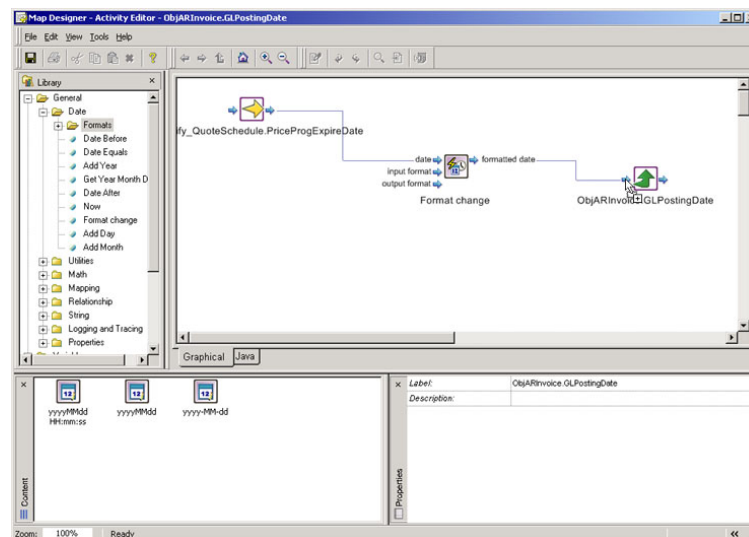


Figure 68. Date Format Change function block with connection links

**Result:** Now we have instructed the Format Change function block to take the input from the attribute ObjClarify\_QuoteSchedule.PriceProgExpireDate, change its date format, and assign the result to the attribute ObjARInvoice.GLPostingDate. However, we still need to let the Format Change function block know what the original date format is and what resulting format we want.

- In our example, if the source attribute ObjClarify\_QuoteSchedule.PriceProgExpireDate is in the date format of yyyyMMDD (that is, 20030227), we can use the predefined Date Format function block yyyyMMdd. Drag-and-drop the yyyyMMdd function block onto the activity canvas and connect the format output of the yyyyMMdd function block to the input format of the Format Change function block.

**Result:** This will specify that the input format of the date is in yyyyMMdd format, as shown in Figure 69.

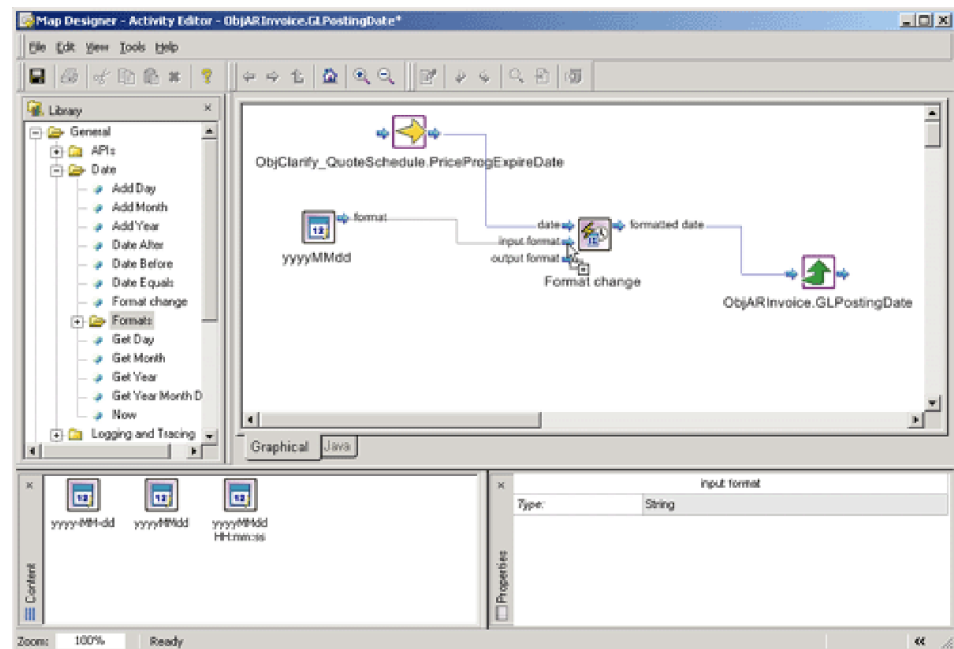


Figure 69. Input Date Format

- Activity Editor provides three predefined Date formats: yyyyMMDD, HH:mm:ss, yyyyMMDD, and yyyy-MM-dd. If the desired date format is not one of the three predefined formats, you can specify the date format you want by using a *Constant*.

**Example:** We want the Format Change function block to change the date format to yyyy.MM.dd G 'at' Hh"mm"ss z. This is not one of the predefined formats, so we will create a New Constant component in the activity canvas by dragging and dropping the *New Constant icon* (located under the System category) from the Content window to the editing canvas. Figure 70 shows the

result of this action.

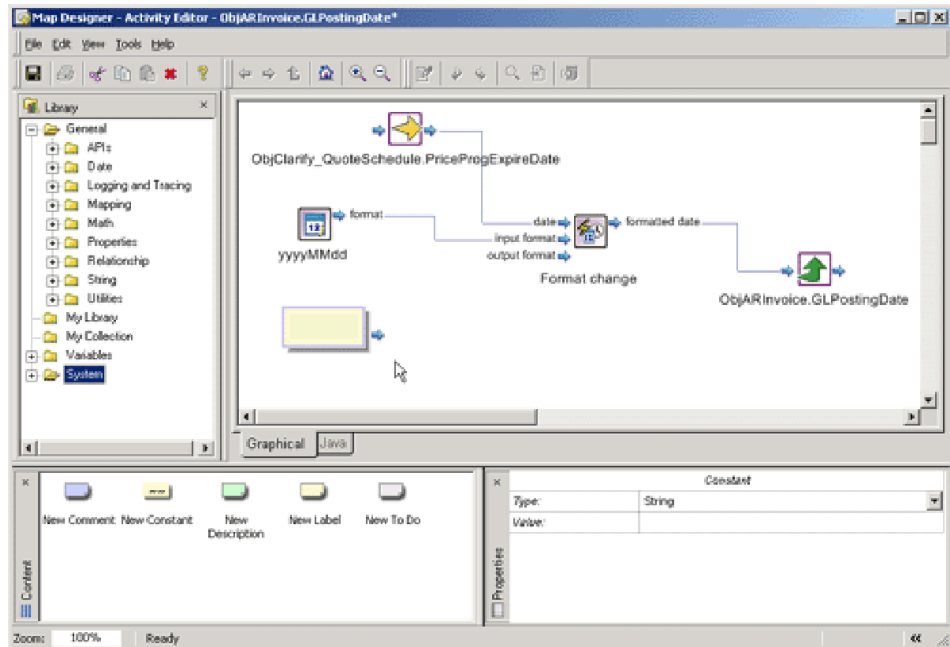


Figure 70. New Constant icon dropped on the activity canvas

8. To specify a constant with the value yyyy.MM.dd G 'at' Hh"mm"ss z, click the editable area of the Constant component in the activity canvas and enter the text yyyy.MM.dd G 'at' Hh"mm"ss z. By default, any Constant component will have the type String (shown in the Properties window when the Constant component is selected). However, you can change the type of the Constant by selecting the Constant and using the combo box in the Properties window. Figure 71 shows the New Constant icon with the text value entered.

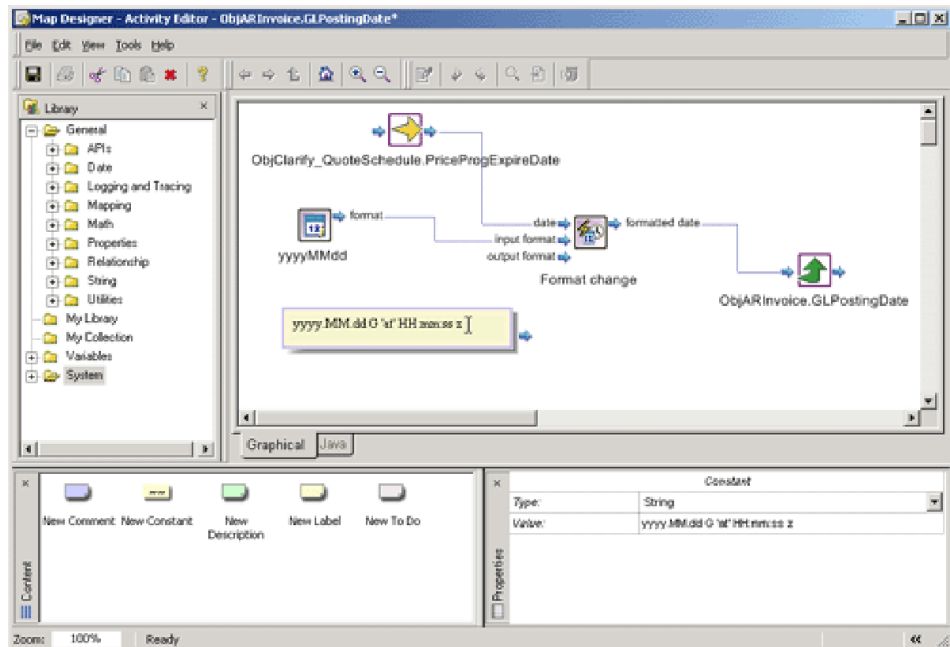


Figure 71. New Constant with text entered

- To continue to specify that we want the output format of the Format Change function block as yyyy.MM.dd G 'at' Hh:mm:ss z, we define a connection link between the Constant component and the output format of the Format Change function block.

**Result:** We have completed the activity definition that will change the date format of the source attribute to a new date format and assign it to the destination attribute.

- To add a comment or description to remind us later what this activity does, we can add a *Description* component to the activity and enter a description.

**Tip:** Use the Context menu in the editing canvas and select Add Description, or drag the *New Description icon* under the System folder in the Content window and drop it onto the editing canvas. Figure 72 shows how to add the Description component using the Context menu.

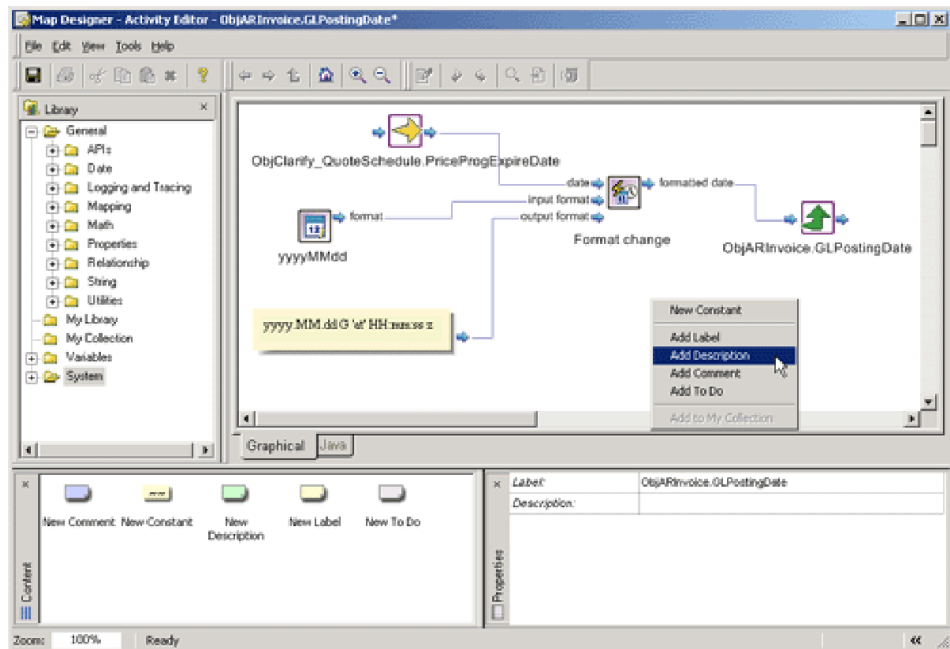


Figure 72. Adding a Description using the Context menu

**Result:** The Description component will be created in the editing canvas.

- Enter the description in the Description component by clicking on the editable area of the component and typing directly into the component. You can resize the Description by clicking and moving the lower right-hand corner of the

Description component. Figure 73 shows adding the Description.

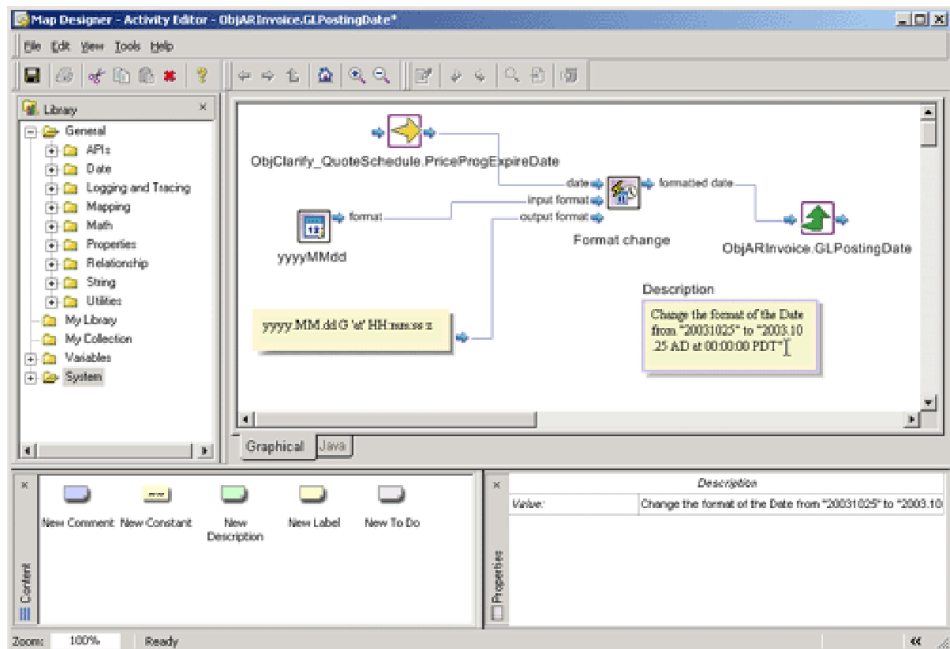


Figure 73. Adding the Description

12. Save the activity by selecting To Project or To File from the File > Save submenu or by clicking the Save Map to Project or Save Map to File button in the Standard toolbar. Figure 74 shows saving the activity.

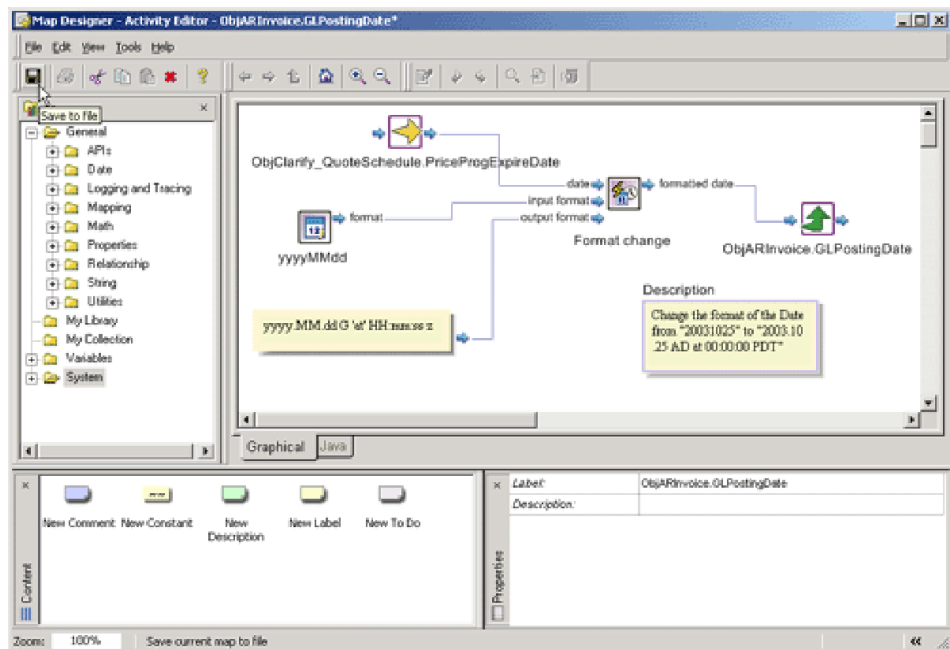


Figure 74. Saving the activity

### Example 3: Using Static Lookup for conversion

The following example illustrates using the *Static Lookup* relationship function block in Activity Editor.

In WebSphere InterChange Server, a static lookup relationship normally consists of two or more relationship tables. For example, consider a system that consists of three end-applications, as shown in Figure 75.

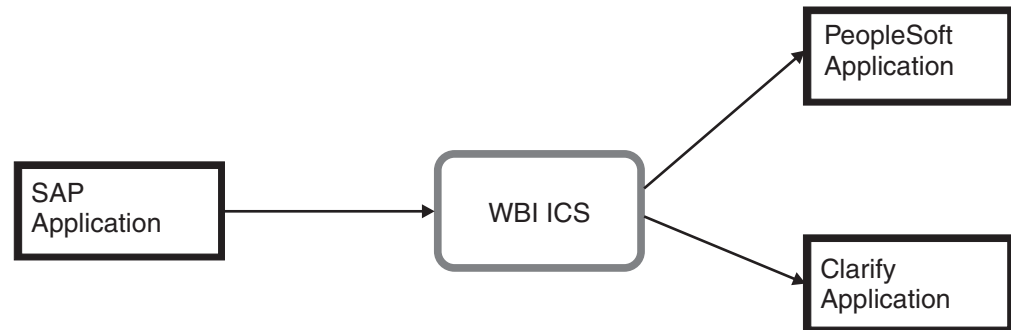


Figure 75. Static Lookup relationship with three end-applications

Each of these three applications has a different representation for "State" information, as shown in Table 59.

Table 59. Application-specific representation of state information

	SAP application	PeopleSoft application	Clarify application
California	CA	01	State1
Washington	WA	02	State2
Hawaii	HI	03	State3
Delaware	DE	04	State4

When state information is sent to the WebSphere business integration system from the SAP application, SAP specified-state code is sent to InterChange Server. But when InterChange Server needs to pass this information to other applications, the state information has to be converted to the format that the target application understands. In order to do this, the system needs a generic representation of the "State" information. With the generic representation, the system can process business logics in a generic, unified manner; and the generic representation will be converted to the application-specific format only when needed.

Thus, in the preceding example, we would create a static lookup relationship for doing this "State" conversion, with the application-specific data as WebSphere business integration-managed participants. With this setup, a generic ID is used to represent the state information in the WebSphere business integration system. Table 60 shows this representation.

Table 60. Generic representation of state information

	Generic ID	SAP application	PeopleSoft application	Clarify application
California	1	CA	01	State1
Washington	2	WA	02	State2
Hawaii	3	HI	03	State3
Delaware	4	DE	04	State4

Application-specific data is converted to the generic ID as it enters the InterChange Server system, and the generic ID is converted to application-specific data as it exits the system. This data conversion is shown in Figure 76.

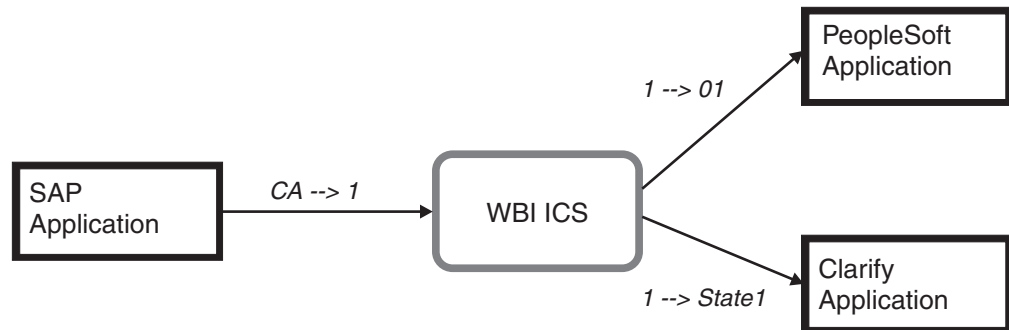


Figure 76. Data conversion from application-specific to generic to application-specific

The ID conversion is usually done in maps that convert application-specific business objects to generic business objects, or vice versa. For example in the SAP-to-Generic map, we would do a static lookup for the data "CA" and convert it to the generic representation that InterChange Server understands, "1". And in the Generic-to-Clarify map, we would instead do a static lookup for the generic data "1" and convert it to "State1". In either map, only one static lookup is required.

Figure 77 shows how to use the Static Lookup function block to convert the SAP-specified state data to the InterChange Server generic state data for processing in InterChange Server.

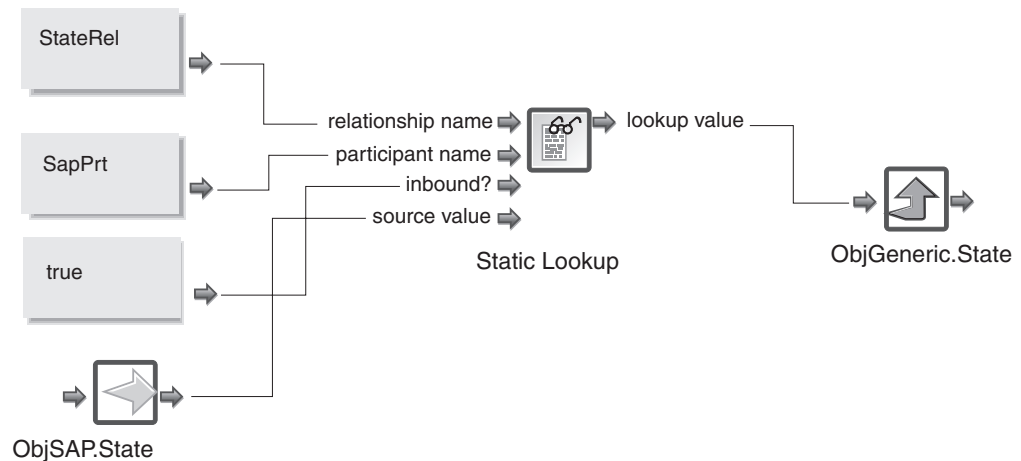


Figure 77. Using static lookup function block to convert SAP-specific state data to InterChange Server-generic state data

Similarly, the Static Lookup function block is used to convert the InterChange Server-generic state data to Clarify-specific state data in the Generic-to-Clarify



map. This is shown in Figure 78.

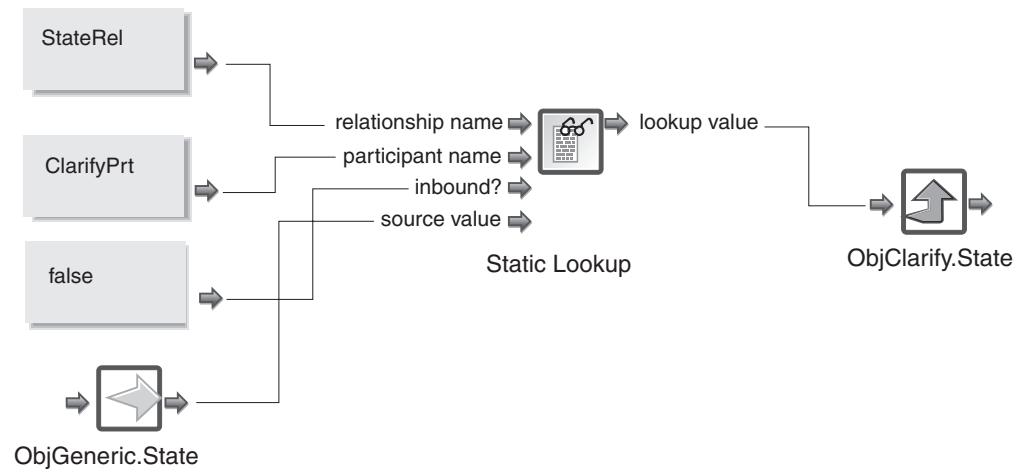


Figure 78. Using static lookup function block to convert InterChange Server-generic state data to Clarify-specific state data

Normally, in a static lookup relationship, we convert application-specific data to generic data, or generic data to application-specific data. In these scenarios, only one Static Lookup function block is used. But in the special cases where you want to directly lookup a name-value pair, then two Static Lookup function blocks are required.

For more information on defining and using static relationships, see Chapter 7, “Creating relationship definitions,” on page 237.

---

## Exporting Web services into Activity Editor

A *Web service* is part of an InterChange Component Library Project in System Manager just like business objects and maps. After the Web service is registered, tested, and verified, its services and methods can be exported as function blocks into Activity Editor for use within maps, like other function blocks.

For information on registering, testing, verifying and exporting a Web service from System Manager into Activity Editor, see the *System Implementation Guide*.

## Using Web services in Activity Editor

After exporting a Web service from System Manager, you need to restart Activity Editor. When Activity Editor opens, the exported Web service is added as a category under My Library. It has the same functionality as other categories in My Library.

Figure 79 shows the Web services category and function blocks in Activity Editor after exporting from System Manager.

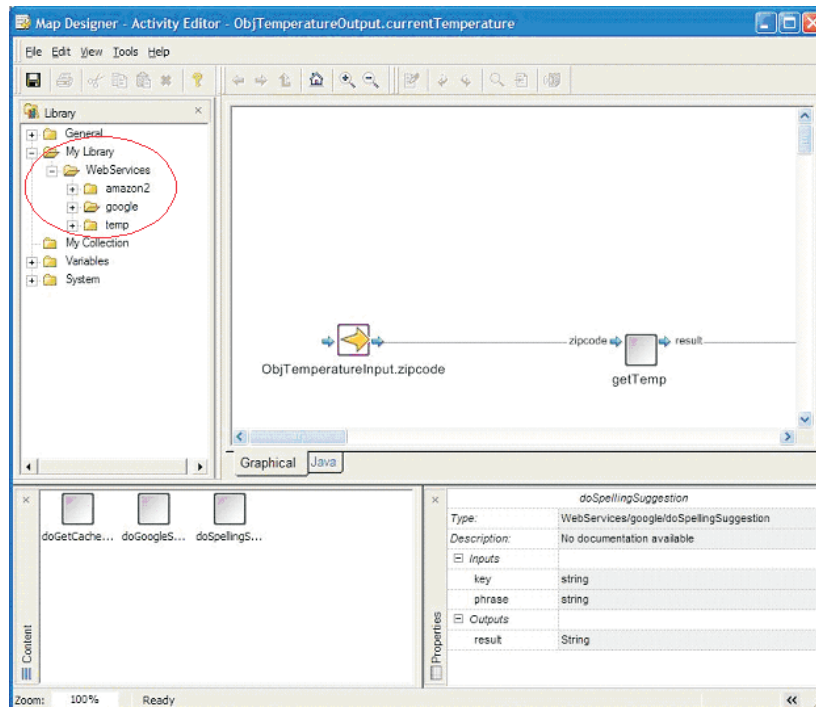


Figure 79. Web services category in Activity Editor

Similar to using other function blocks in Activity Editor, using any of the Web services function blocks is a matter of dragging and dropping the function blocks and connecting the inputs and outputs. For information on using Activity Editor, see “Working with activity definitions” on page 112.

## Example of using a Web service in a map

The following example illustrates how to invoke a Web service using Activity Editor to change a source attribute’s zip code to the temperature for the city and assign the change to the destination attribute.

Perform the following steps:

1. From the Diagram tab of Map Designer, create a custom transformation by dragging the source business object attribute `ObjTemperatureInput.zipcode` onto the destination business object attribute `ObjTemperatureOutput.currentTemperature`. Then click the icon of the Custom transformation rule to launch Activity Editor.

Figure 80 shows the custom transformation.

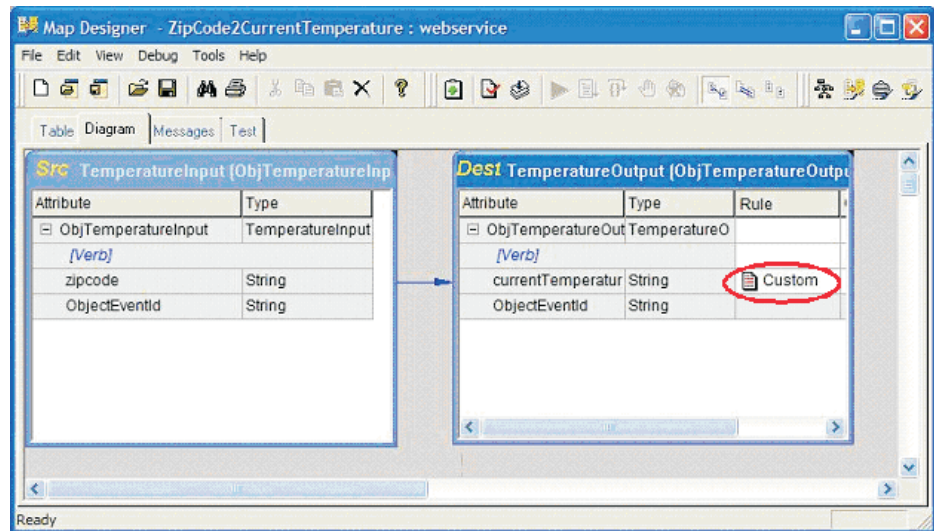


Figure 80. Creating the custom transformation

2. Select the Web services category in the Library window to show the function blocks available in that category, as icons, in the Content window.
3. Drag and drop the Web services getTemp function block from the Content window onto the editing canvas.
4. Connect the output port of the icon for the source business object attribute ObjTemperatureInput.zipcode to the input port "zipcode" of the getTemp function block; and connect the output port "result" of the getTemp function block to the input port of the icon of the destination business object attribute ObjTemperatureOutput.currentTemperature.

Figure 81 shows the connected inputs and outputs of the getTemp function block.

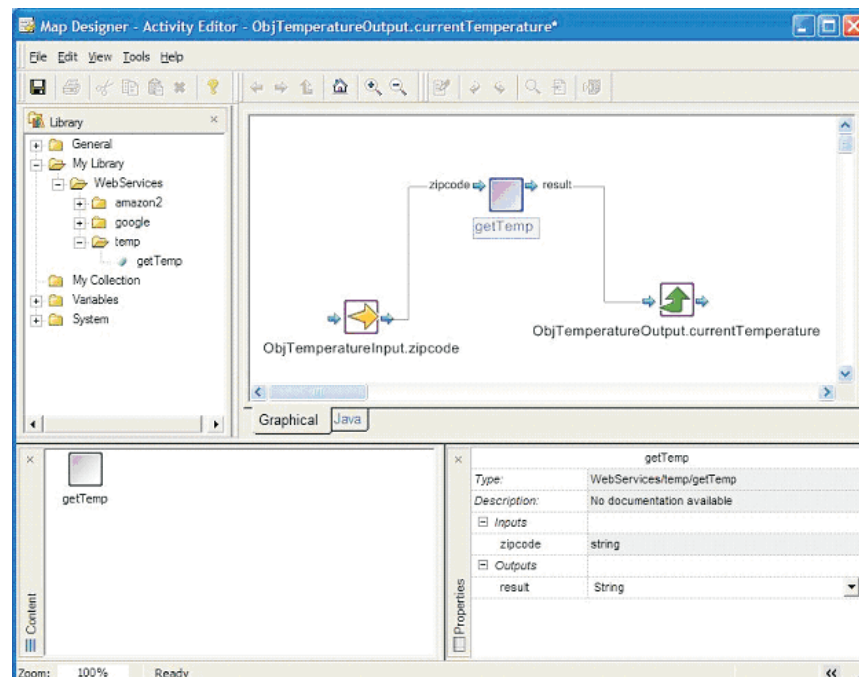


Figure 81. Connecting the inputs and outputs

5. Save the activity template and map.
6. Switch to Test view in Map Designer. Enter a valid zip code in the Source-zipcode field. Click Debug Map. You can choose to deploy the map and business objects to the server if you have not already done so.

**Result:** After the test run is finished, you will see the current temperature for the zip code in the destination business object.

Figure 82 shows how the zip code 94010 of the source business object attribute has been transformed to 57 degrees for the destination business object attribute.

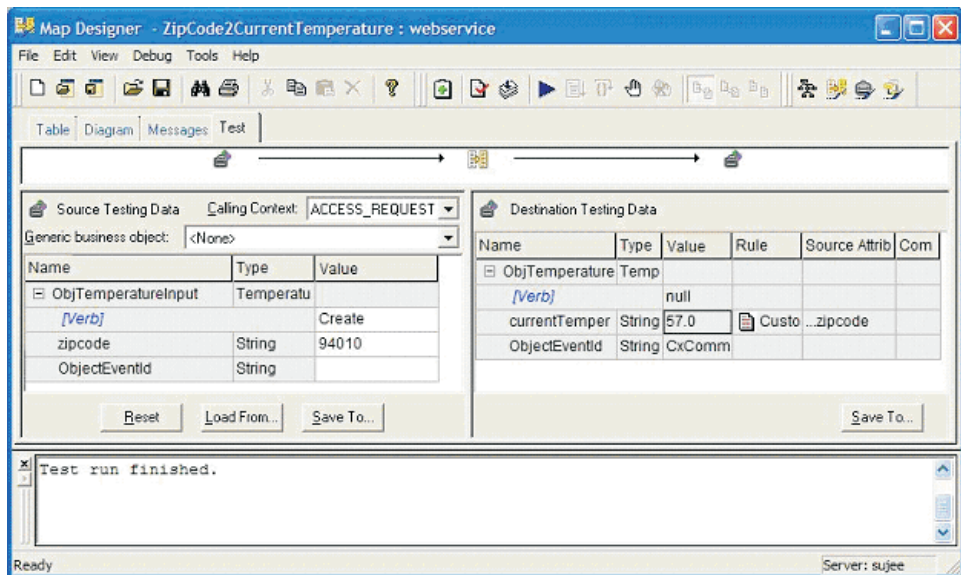


Figure 82. Test view results

## Using bidirectional functionality in Activity Editor

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

An adapter that supports bidirectional languages. To determine if your adapter supports bidirectional languages, see your adapter’s user’s guide.

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

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

- Only accepting input from sources that enforce the same bidirectional format as the WebSphere Business Integration Server such as the adapters that are already enabled with this support.

- Enabling the connectors to this map to enforce the correct bidirectional format (see "Enabling connectors for bidirectional languages" in the *Collaboration Development Guide*).
- Using the APIs in the CwBidiEngine class to transform all data into a consistent bidirectional format (see Chapter 12, "CwBidiEngine class," on page 361).

InterChange Server automatically enables BiDi functionality with the following nine connectors: JText, JDBC, Email, XML, MQ Series, SAP, PS, Web services, Lotus Domino. Therefore, when data in Windows BiDi format utilizes these enabled connectors in Web service no special configuration is needed.

In the event that Web service operates with BiDi data that is not in Windows BiDi format, two results are possible. The first result is that the connection to such a service might fail all together. The second result is that the BiDi data is in a format different from Common Windows Bidirectional Format (CWBF) that gives unpredictable results in data processing because the data is being compared against data in the CWBF format. In other words, identical information is being held in different BiDi formats. To rectify these potential situations, you need to follow these steps for Web service deployment inside Activity Editor.

## Steps for deploying Bidi API in Web service

Follow these steps to deploy BiDi in Web service:

1. Register the Web service.  
For information on registering, testing, verifying and exporting a Web service see the *System Implementation Guide*.
2. Test the Web service using BiDi data to determine the BiDi format standard.
3. Export the Web service into Activity Editor.
4. Design the data flow using Activity Editor. Figure 83 on page 164 shows an example of a BiDi design process.

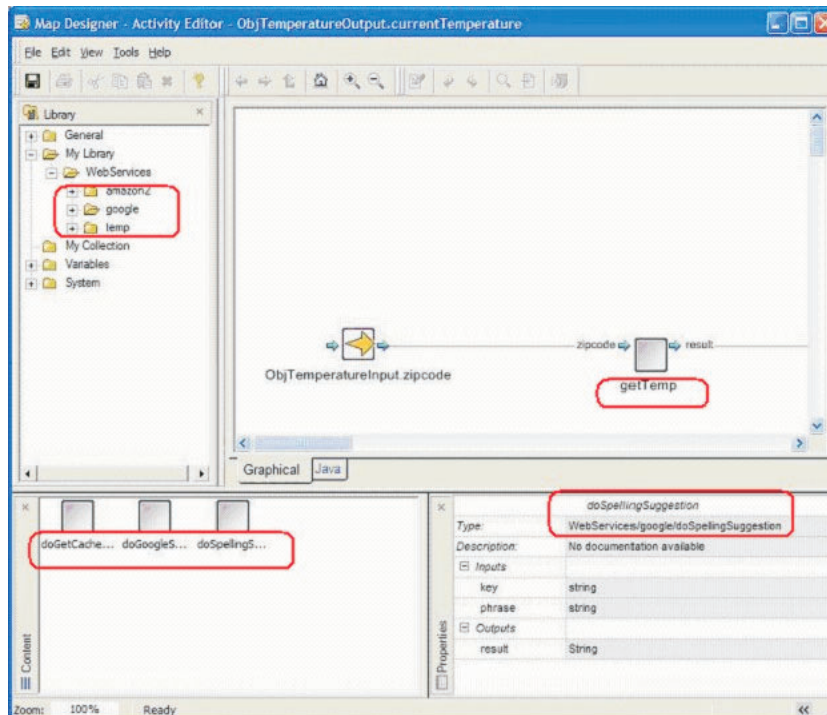


Figure 83. Activity Editor screen with BiDi elements circled in red

5. Add calls to BiDi API inside the generated code just before the data is sent to the Web service and just after the reply is received to preserve the BiDi data consistency.

**Note:** For more information on the BiDi transformations of BO instance content used in the BiDi API, see Chapter 12, “CwBidiEngine class,” on page 361.

---

## Importing Java packages and other custom code

Map Designer provides two ways to import Java packages and other custom code:

- “Importing Jar libraries as activity function blocks” on page 164
- “Importing through the Map Properties dialog” on page 166

A description of each method follows.

### Importing Jar libraries as activity function blocks

In addition to using the standard function blocks that Activity Editor provides, Map Designer allows you to import your own Java library for use as function blocks in 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 Activity Editor.

#### Steps for importing Jar libraries as activity function blocks

**Before you begin:** You need to export your Java classes into a .jar file.

Perform the following steps to import a Jar library into Activity Editor:

1. In System Manager, open the Activity Settings view by clicking Window > Show View > Other and selecting Activity Settings from the category System Manager.
2. Right-click BuildBlock Libraries and select Add Library. Figure 84 shows the Activity Settings view for adding a custom Jar library.

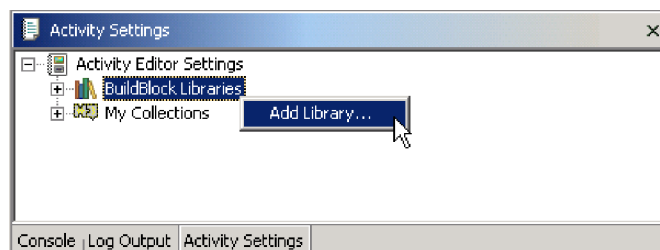


Figure 84. Activity Settings view

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 you import your custom .jar files into Activity Settings, when your maps and collaboration template compile in System Manager, the custom .jar file will automatically be included into the compile CLASSPATH. To prepare



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 167.

4. Restart Map Designer.

**Rule:** After you change any settings in the Activity Settings view, you must restart Map Designer for the change to take effect in Activity Editor.

**Result:** When you open 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 standard function blocks.

### Customizing display settings of custom Jar libraries

You can customize the display settings of the function blocks imported into Activity Editor, such as its name and icon, by changing the custom Jar library’s properties. Perform the following steps to do this:

- 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, and enabling this option can better organize your My Library subtree in Activity Editor.

Figure 85 shows the dialog for customizing the Jar library display.

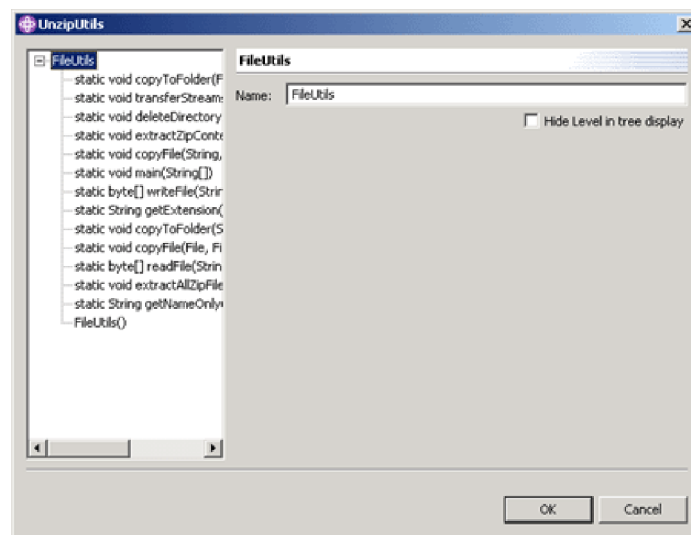


Figure 85. Properties dialog for customizing Jar library display

For those Java methods used as function blocks in 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:** If you choose to import 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 86 shows the Properties dialog for customizing the Jar library function block display.

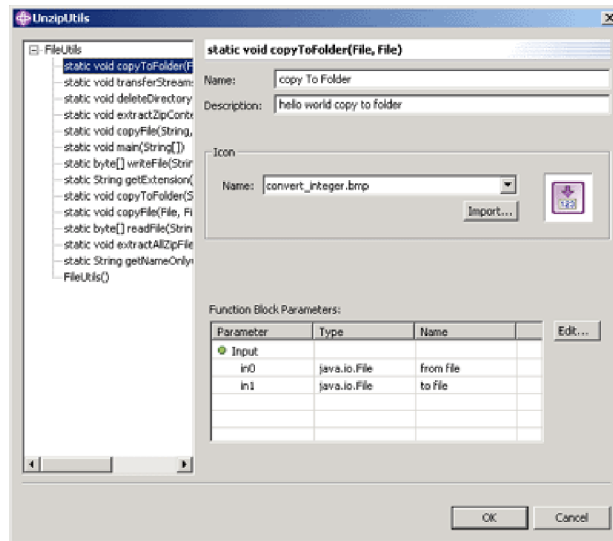


Figure 86. Properties dialog for customizing Jar library function block display

**Rule:** After you change any settings in the Activity Settings view, you must restart Map Designer for the change to take effect in Activity Editor.

**Note:** You can import and export activity settings using the Import/Export wizard in the Eclipse Workbench. For more information, see the documentation for the Eclipse Workbench.

## Importing through the Map Properties dialog

Map Designer automatically imports the Java packages necessary to execute a map. If you write transformation code that uses methods from another Java package, or if you use the Map Utilities (MapUtils) package, you need to import the package into the map. You can also write your own custom Java code outside of Map Designer and import your code into a map for use within transformation code.

**Note:** The version of the Java Development Kit (JDK) that is appropriate for the IBM WebSphere InterChange Server release must be installed for map development.

**Attention:** Map Designer cannot debug or verify the logic of any code imported into a map. Consequently, the map developer is responsible for all errors and exceptions caused by imported code.



## Steps for importing Java packages or other custom code

To import a Java package or other custom code:

1. Display the General tab of the Map Properties dialog, as follows:  
From the Edit menu, select Map Properties. For information on other ways to display the Map Properties dialog, see “Specifying map property information” on page 58. The Map Properties dialog box appears (see Figure 91 on page 189).
2. Enter the import statements in the Map file declaration block. You can also enter other Java statements in the Map local declaration block.

**Note:** When you compile the map, the Java compiler looks for imported packages in the directory defined by the CLASSPATH environment variable. If you import a package into a map and deploy the map at another InterChange Server system before compiling it, make sure you also deliver the imported package with the map.

For requirements for compiling a map with System Manager, see “Compiling a map” on page 84.

All statements execute at the beginning of the map, *before* any transformation steps execute.

3. Click OK to close the Map Properties dialog box.

## Steps for importing map utilities

To use the Map Utilities package, you must import it into the map by performing the following steps:

1. Ensure the CollabUtils.jar file is in the <ProductDir>\lib directory.
2. Ensure the start\_server.bat (or CWSharedEnv.sh) file contains a reference to the CollabUtils.jar file.

**Note:** Steps 1 and 2 are needed for the server compile.

3. Modify the Compiler Classpath preference for compiling the maps and collaboration templates from System Manager:
  - a. In System Manager, select Window > Preferences to open the Preference dialog.
  - b. Expand System Manager Preferences, and select Compiler.
  - c. In the Compiler preference page, click New and select the .jar file to be included in the compile CLASSPATH for maps and collaboration templates.
4. From Map Designer, open the map that needs to use the Map Utilities package.
5. Display the Map Properties dialog.
6. In the Map Properties dialog box, type the following import statement in the Map file declaration block:

```
import com.crossworlds.MapUtils.*;
```
7. Click OK to close the Map Properties dialog box.

## Importing third-party classes to InterChange Server

If the imported classes are in a third-party package rather than in the JDK, in order to set up the server compile, you must add them to the path of the imported classes in the JCLASSES variable.

**Recommendation:** You should 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 performing the following steps:

1. Create a new map 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="%ProductDir%\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 do this, perform the following steps:

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 the following:

- 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\, 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.`

---

## Using variables

A variable is a placeholder for a value in the Java code. This section provides the following information about using variables in transformation code:

- “Using generated business object variables and attributes”
- “Creating temporary variables” on page 171

### Using generated business object variables and attributes

This section provides information about generating business object variables for the source and destination business objects. When you add a business object to the map, Map Designer automatically generates the following:

- An instance name

The instance name that Map Designer generates is a system-declared local variable that you can use to refer to this business object in the mapping code. It is prefixed with the letters `Obj`, which is followed by the name of the business object definition.

**Example:** If you add `Customer` to the map, its instance name is `ObjCustomer`. Map Designer generates an instance name for both the source and destination business objects.

When you write code in Activity Editor, you use the instance name to refer to the business object and its attributes.

- An index for the business object within a business object array (if the business object is multiple-cardinality)

The business object index represents the order of this source or destination business object. The index number of the first source and destination business objects in a map is zero. Additional business objects take the next available index number, such as 1, 2, 3, and so on.

When the map is executed, the index number represents the position of the business object in the array that is passed into the map (source business objects) or returned by the map (destination business objects).

Map Designer displays this information in the following locations:

- In the Business Objects tab of the Map Properties dialog

Right-click the title bar of the business object window and select Properties from the Context menu. The Map Properties dialog appear with the Business Objects tab displaying and the selected business object highlighted in the list. This tab displays both the instance name and its index within the business object array (if the business object is multiple cardinality).

- In the Table tab—in the business object pane
- In the Diagram tab—in the title bar of the business object window in the following format:

The title bar displays the instance name for the business object.

**Note:** You can specify whether Map Designer displays the names of the variables for the source and destination business objects with the option *Defining Map: show business object instance name*. By default, this option is enabled and Map Designer displays these variable names (*ObjBusObj*) in both the Table and Diagram tabs. When the option is disabled, Map Designer only displays the names of the source and destination business objects. You can change the setting of this option on the General tab of the Preferences dialog. For more information, see “Specifying General Preferences” on page 22.

## Steps for modifying business object variables

You can modify these business object variables from the Business Objects tab of the Map Properties dialog (see Figure 87).

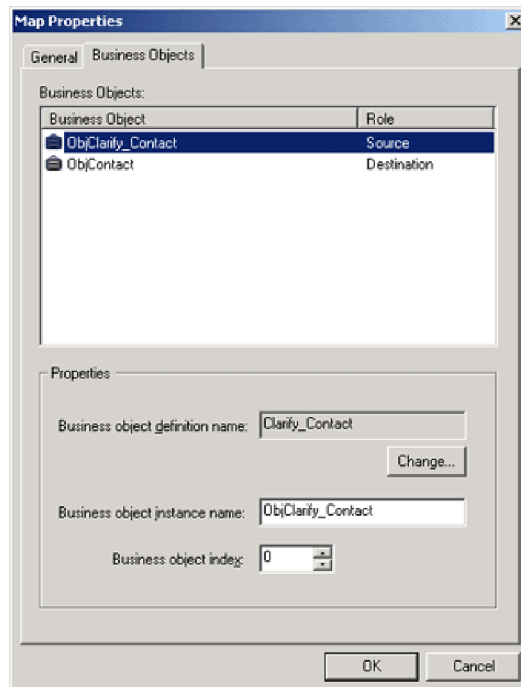


Figure 87. Business Objects Tab of the Map Properties Dialog

To change the business object type of the source or destination business object in the map in the Map Properties dialog, perform the following steps:

1. Open the map.
2. Display the Business Objects tab of the Map Properties dialog in any of the following ways:
  - From the Edit menu, select Map Properties.
  - From the Diagram tab, right-click the business object window and select Properties from the Context menu.

**Result:** The General tab of the Map Properties dialog box appears. Click the Business Objects tab.

For information on other ways to display the Map Properties dialog, see “Specifying map property information” on page 58..

3. Select the business object type you want to change.
4. Click the Change push- button under Business Object Type.
5. Select the new type.

6. Click OK to close the Select Business Object dialog.
7. Click OK to close the Map Properties dialog.

**Note:** Invalid transformation rules will be deleted.

## Referencing business object attributes

Use the business object variables that Map Designer generates to reference business objects and their attributes, as follows:

- To reference attributes in a source or destination business object, use the business object name followed by the attribute name, with a period (.) as a separator:

**Example:**

```
ObjBusObjName.AttrName
```

- To reference attributes in a child business object, use the child business object name and the child attribute name.

**Example:** The following example sets the value of the `OrigHireDate` attribute in `ObjPsft_Employee` to the `HireDate` attribute of `EmployeeHR_Misc`, which is a child of `ObjEmployee`:

```
ObjPsft_Employee.set("OrigHireDate",
    ObjEmployee.getString("EmployeeHR_Misc.HireDate"));
```

- If the child business object has a cardinality of *n* (meaning there can be more than one instance of the child associated with the parent), you must supply an index number for the child business object.

**Example:** The following example sets the value of the `TimeZone` attribute of `Address`, which is a multiple-cardinality child of `ObjCustomer`:

```
ObjCustomer.set("Address[0].TimeZone",
    ObjSAP_Customer.getString("TimeZone"));
```

## Creating temporary variables

Map Designer lets you create temporary variables that can be accessed in transformation steps throughout the map; that is, temporary variables are global to the map. For example, you can calculate a value in one transformation step, store it in a temporary variable, and reference the variable in another transformation step. This is especially useful if a certain calculation is performed repeatedly; you can perform the calculation once, store the result in a temporary variable, and retrieve the value as needed (for example, with a Move transformation).

### Steps for creating temporary business object variables

Temporary variables are defined within a temporary business object. Perform the following steps to create a temporary business object variable:

1. Select Add Business Object from the Edit menu.

**Result:** The General tab of the Add Business Object Properties dialog box appears.

For information on other ways to display the Add Business Object dialog, see “Steps for specifying business objects from the Add Business Object dialog” on page 34.

2. Click the Temporary tab. This is where you define the temporary variables. Figure 88 shows the Temporary tab of the Add Business Object dialog. In the Name field appears the temporary business object’s name, which Map Designer

has generated. The first generated name is ObjTemporary. This field is read-only.

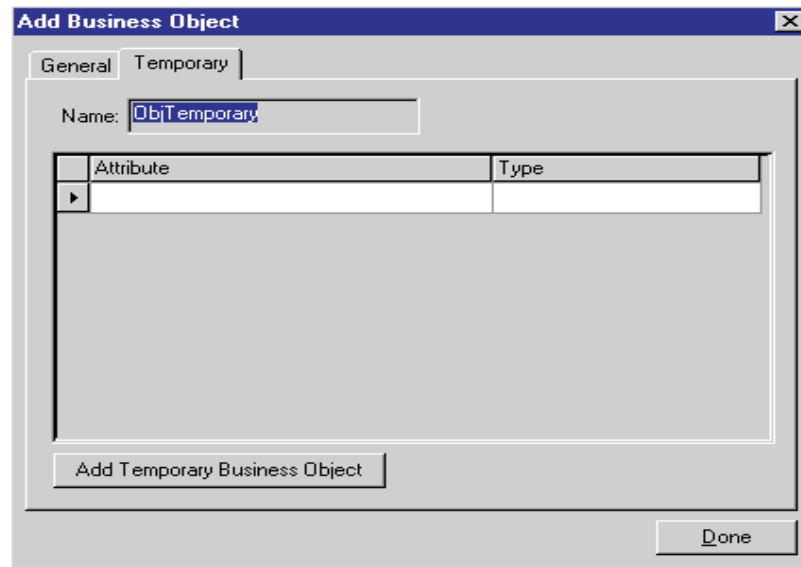


Figure 88. Temporary tab of the Add Business Object dialog

3. Click in the Attribute field.

**Result:** A new row appears in the variables table. Enter the name of the temporary variable.

**Note:** Do *not* create two temporary variables with the same name.

4. Click the Type field and select the temporary variable's data type from the pull-down list.

**Note:** To be compatible with the InterChange Server data type scheme, all temporary variables have an internal type String. The data type specified in the Add Business Object dialog will affect only how the variable is initialized. If you want to write custom Java code to assign values to the temporary variable, the value has to be converted to a String first.

5. Repeat steps 3 and 4 for each of the temporary variables you need in the map.
6. Click the Add Temporary Business Object button.
7. You can either define another temporary business object or click Done to finish.

### Steps for using temporary business object variables in transformation steps

Use the temporary variable in a transformation step in this way:

- In the Diagram tab:
  1. Click the row header (left-most) column of the temporary attribute.
  2. Copy the variable value to an attribute by holding the Ctrl key and dragging the variable onto the attribute.
- In the Activity Editor Java tab, use the variable name in the transformation step for an attribute.

**Important:** Because a temporary variable is a global variable, you must explicitly initialize a temporary variable to null when using the Map Instance Reuse option. Otherwise, the value of the temporary variable from a previous execution of the map instance can incorrectly be used as the

value of the temporary variable in subsequent executions of the same map. When you do *not* use the Map Instance Reuse option, the InterChange Server system automatically initializes temporary variables between separate invocations of the map.

**Result:** Once Map Designer creates the temporary business object, this business object appears in the Table and Diagram tabs with the map's other business objects, as follows:

- In the Table tab:
  - The business object pane adds a new area for the temporary business object. Right-click the name of the temporary business object to open a Context menu with options to edit and delete this business object.
  - The temporary business object and its attributes appear in the combo boxes of the Source Attribute and Dest. Attribute columns in the attribute transformation table.
- In the Diagram tab, the map workspace adds a new business object window for the temporary business object.

This Temporary business object window has many of the same characteristics as a business object window. Variables you create appear in the variables table just like attributes in a business object. This business object window provides a Rule and Comment column where you can add the temporary variable's transformation code and comment, respectively.

You can right-click in the title bar of the Temporary business object window to get a Context menu that provides options to edit and delete this business object, as well as its properties. Specify a value for the variable in one of the following ways:

- To enter code that returns the value of the variable, double-click in the Rule column, select the appropriate transformation rule, and click Edit Code to enter the code in Activity Editor.
- To copy a value from a business object attribute to a variable, hold down the Ctrl key and drag the attribute onto the variable name. You can also split and join attributes into a variable.

**Note:** A temporary business object also appears in the Business Object tab of the Map Properties dialog.

## Declaring variables

**Tips:** Keep the following tips in mind when declaring variables:

- When creating a variable local to the current attribute (not visible to all other attributes), declare it at the top of the current attribute's transformation step (in Activity Editor).
- When creating a variable global to the current map (visible to all attributes), declare it in the Map Local Declaration block section of the General tab in the Map Properties dialog. When writing code to assign values to these variables, do this at the top of Activity Editor in the first attribute of the destination object (as specified by the execution order).

---

## More attribute transformation methods

You can perform attribute transformations interactively in the following ways:

- Using *only* Map Designer—create one of the standard transformations.



Table 14 on page 36 lists the standard transformations for which Map Designer can generate code.

- Using a combination of Map Designer and Activity Editor to modify and enhance the code—create a Custom transformation.

You can create customized transformations in either of the following ways:

- By creating a standard transformation and opening Activity Editor to change the generated code. After you customize a standard transformation, Map Designer displays the type of the transformation in blue italicized font in the transformation rule column.
- By creating a Custom transformation and opening Activity Editor to define the transformation. After you create a Custom transformation, Map Designer displays the keyword `Custom` in black font in the transformation rule column. For more information, see “Creating a Custom transformation” on page 47.

This section describes how to implement the following kinds of custom transformations:

- “Content-based logic”
- “Date formatting” on page 179
- “Using Expression Builder for string transformations” on page 182

## Content-based logic

```
Customer.CustomerStatus = 'Inactive'  
if SAP_CustomerMaster.DeleteInd = 'X'.
```

```
Otherwise, CustomerStatus = 'Active'.
```

You can create a Custom transformation and write the entire piece of code to implement the content-based logic yourself. However, a better approach is to start by creating a Move transformation between `DeleteInd` (in the `SAP_CustomerMaster` object) and `CustomerStatus` (see “Copying a source attribute to a destination attribute” on page 38 for the procedure to move attributes).

As a result, Map Designer generates the move-transformation, as shown in the following sample code:

```
{  
  Object _cw_CpBTBSourceValue = null;  
  
  //  
  // RETRIEVE SOURCE  
  // -----  
  //  
  // Retrieve the source value from the source business object and  
  // place it in a local variable for code safety.  
  //  
  _cw_CpBTBSourceValue = ObjSAP_CustomerMaster.get("DeleteInd");  
  
  //  
  // SET DESTINATION  
  // -----  
  //  
  // Put the source value into the destination business object  
  // attribute.  
  //  
  {  
    Object _cw_SetSrcVal = _cw_CpBTBSourceValue;  
    BusObj _cw_SetDestBusObj = ObjCustomer;  
    String _cw_SetDestAttr = "CustomerStatus";
```



```

//
// Set the destination value only if neither
// source nor destination is null.
//
if ((_cw_SetSrcVal != null) && (_cw_SetDestBusObj != null))
{
    if (dataValidationLevel >= 1)
    {
        if (!_cw_SetDestBusObj.validData(_cw_SetDestAttr, _cw_SetSrcVal))
        {
            String warningMessage =
                "Invalid data encountered when attempting to set the value of the
                \'_cw_SetDestAttr\' attribute of BusObj \'_cw_SetDestBusObj\' while running
                map \'\' + getName() + "\'. The invalid value was \'\' + _cw_SetSrcVal
                + "\'.";

            //
            // Log a warning about this failure.
            //
            logWarning(warningMessage);
            if (failOnInvalidData)
            {
                //
                // Fail the map execution with a warning message.
                //
                throw new MapFailureException(warningMessage);
            }
        }
    }
}
// SECTION THAT NEEDS TO BE UPDATED WITH THE LOGIC

if (_cw_SetSrcVal != null)
{
    if (_cw_SetSrcVal instanceof BusObj)
    {
        //
        // Since BusObjs are not immutable, we need to make
        // a copy of the source object before actually
        // putting it into the destination attribute.
        //
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            ((BusObj)_cw_SetSrcVal).duplicate());
    }
    else if (_cw_SetSrcVal instanceof BusObjArray)
    {
        //
        // Since BusObjs are not immutable, we need to make
        // a copy of the source object before actually
        // putting it into the destination attribute.
        //
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            ((BusObjArray)_cw_SetSrcVal).duplicate());
    }
    else
    {
        //
        // Since our version of simple data types are immutable in
        // Java (Strings included), we do not have to make a copy
        // of the source value here.
        //
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, _cw_SetSrcVal);
    }
}
}
}

// HERE IS THE MODIFIED CODE

```

```

if (_cw_SetSrcVal != null)
{
    if (((String)_cw_SetSrcVal).equals("X"))
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, "Inactive");
    else
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, "Active");
}
}
}
}

```

Select Save from Activity Editor File menu to save the changes.

### Verb-based logic

Customer.CustomerStatus = 'Inactive' if Verb = 'Create'. Otherwise, CustomerStatus = 'Active'.

Follow the procedure in “Content-based logic” on page 174, but use the following conditional statement to replace the section of the generated code:

```

// HERE IS THE MODIFIED CODE
if (_cw_SetSrcVal != null)
{
    if (ObjSAP_CustomerMaster.getVerb() .equalsIgnoreCase("Create"))
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, "Inactive");
    else
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, "Active");
}

```

### Providing default values if source data is missing

Map SAP\_CustomerMaster.State into Customer.CustomerAddress.State. If SAP’s state is missing, default to CA.

Notice that before the destination attribute is set to the source value, the code checks if the source attribute is not equal to null.

**Example:** In this example, if the source data is missing, set the destination attribute to a default value.

Start by moving SAP\_CustomerMaster.State into Customer.CustomerAddress.State. Change the condition statement to the following:

```

// HERE IS THE MODIFIED CODE
if (_cw_SetSrcVal != null)
    _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, _cw_SetSrcVal);
else
    _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, "CA");
_cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, "CA");

```

**Tip:** If you started the coding by copying the source attribute into the destination attribute, the line of code that sets the default value has to be typed twice. This is because the code that is generated when the source attribute is moved into the destination attribute checks twice that the source attribute is not null. Thus, you must enter the default value.

### Logic based on calling context

When you need to check for the value of the calling context, use the built-in variable strInitiator, which is of type String.

**Example:** To check if the calling context is EVENT\_DELIVERY, use the following statement:

```

if (strInitiator.equals(MapExeContext.EVENT_DELIVERY))
//rest of the code

```

### Forcing a map to fail if the source data is missing

Map SAP\_CustomerMaster.State into Customer.CustomerAddress.State. If SAP's state is missing, stop the map from executing.

Start by moving SAP\_CustomerMaster.State into Customer.CustomerAddress.State. Then add one more if() statement to check if the State attribute for SAP is equal to null. The generated code looks like the following:

```

{
Object _cw_CpBTBSourceValue = null;

//
// RETRIEVE SOURCE
// -----
//
// Retrieve the source value from the source business object and
// place it in a local variable for code safety.
//
_cw_CpBTBSourceValue = ObjSAP_CustomerMaster.get("State");

//
// SET DESTINATION
// -----
//
// Put the source value into the destination business object
// attribute.
//
{
Object _cw_SetSrcVal = _cw_CpBTBSourceValue;
BusObj _cw_SetDestBusObj = ObjCustomer;
String _cw_SetDestAttr = "CustomerAddress.State";
// New code
if (_cw_SetSrcVal == null)
{
String errorMessage = "Data in the state attribute is missing";
logError(errorMessage);
//
// Fail the map execution with a warning message.
//
throw new MapFailureException(errorMessage);
}
// End of new code

//
// Set the destination value only if neither
// source nor destination is null.
//
else if ((_cw_SetSrcVal != null) && (_cw_SetDestBusObj != null))
{
if (dataValidationLevel >= 1)
{
if (!ObjCustomer.validData("CustomerAddress.State", _cw_SetSrcVal))
{
String warningMessage =
"Invalid data encountered when attempting to set the value of
the \"CustomerAddress.State\" attribute of BusObj \"ObjCustomer\"
while running map \"' + getName() + \"'. The invalid value
was \"' + _cw_SetSrcVal + \"'.";

//
// Log a warning about this failure.
//
logWarning(warningMessage);

```

```

        if (failOnInvalidData)
        {
            //
            // Fail the map execution with a warning message.
            //
            throw new MapFailureException(warningMessage);
        }
    }
}

if (_cw_SetSrcVal != null)
    _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr, _cw_SetSrcVal);
}
}
}

```

If the map is run and the State attribute is not set, an error message is produced in both a dialog window and on the server screen. The message on the server screen is similar to the following:

```

[1999/09/22 17:58:51.008] [Server]
Sub_SaCwCustomerMaster: Error Data
in the state attribute is missing

```

The map stops executing.

**Tip:** Use the following code to display the actual error message generated by the system on a map failure:

```

try
{
    // your code
}
catch (Exception e)
{
    throw new MapFailureException(e.toString());
}

```

### Logging messages from a message file

Notice how the example in “Forcing a map to fail if the source data is missing” on page 177 uses the `logError()` method to display the error message on the screen. You can use messages provided in the generic messages file, `CWMapMessages.txt`, which is stored in `\DLMs\messages`. See Appendix A, “Message files,” on page 497, for more information about message files.

`CWMapMessages.txt` has the following format:

```

# critical data is missing error
10
Data in the {1} attribute is missing. Map execution stopped.
# Another error
11
Another error message

```

Notice that instead of `{1}` you can use the word `State` to correspond directly to the error that needs to be displayed. If implemented, however, the message is no longer generic. If another attribute also has critical data, you need another message in the message file specific to that particular attribute.

**Note:** Do *not* modify the `CwMapMessages.txt` file. If you need to write your own messages, enter them in the Messages tab of Map Designer, which creates a map-specific error message file named `mapName_locale.txt` (where `mapName`

is the same name as the map), for example, mapName\_en\_US.txt. The server saves this message file in the directory \DLMS\messages after deployment.

To display message #10, use the following code:

```
logWarning(10, "State");
```

The word State replaces {1} in the message text.

The following message is displayed in the InterChange Server log file:

```
[1999/09/23 10:17:43.648] [Server]
Sub_SaCwCustomerMaster:
Warning 10: Data in the State attribute is missing.
Map execution stopped.
```

## Date formatting

The Mapping API provides methods for date formatting in the DtpDate class. Table 61 summarizes the date-formatting methods.

Table 61. Date-Formatting Methods of the DtpDate Class

Date Formatting	DtpDate Method
Getting the month name from a date	getMonth(), getShortMonth()
Getting the month value from a date	getIntMonth(), getNumericMonth()
Getting the day of the month	getDayOfMonth(), getIntDay()
Getting the day of the week	getDayOfWeek(), getIntDayOfWeek()
Getting the year from a date	getYear(), getIntYear()
Getting the hour value	getHours()
Getting the minutes value from a date	getMinutes(), getIntMinutes()
Getting the seconds value from a date	getSeconds(), getIntSeconds()
Getting the number of milliseconds in the date	getMSSince1970()
Getting the earliest date from a list	getMinDate(), getMinDateB0()
Getting the most recent date from a list	getMaxDate(), getMaxDateB0()
Parsing the date according to a specified format	DtpDate()
Getting the date in a specified or default format	toString()
Reformatting a date to the IBM-generic date format	getCWDate()
Adding days to date	addDays()
Adding weekdays to date	addWeekdays()
Adding years to date	addYears()
Calculating days between dates	calcDays()
Calculating weekdays between dates	calcWeekdays()
Comparing dates	after(), before()
Using full month names	get12MonthNames(), set12MonthNames(), set12MonthNamesToDefault()
Using short month names	get12ShortMonthNames(), set12ShortMonthNames(), set12ShortMonthNamesToDefault()
Using weekday names	get7DayNames(), set7DayNames(), set7DayNamesToDefault()

**Tip:** Always catch DtpDateException. This guarantees that if the date format is invalid, a message is sent to the InterChange Server log.

## Using the generic date format

IBM uses the following date format in its generic business objects:

```
YYYYMMDD HHMMSS
```

This format is called the *generic date format*.

**Steps for converting to generic date format:** To convert an application-specific date to this generic format, use the `getCWDate()` method of the `DtpDate` class.

**Example:** To map the SAP date attribute into a generic date, perform a Copy of the source attribute (the SAP date string) into the destination attribute (the generic date string):

1. Create a `DtpDate` object to hold the generic date.  
Copy the source attribute (the SAP date string) into a new `DtpDate` object (the generic date string) by parsing the SAP date string (YYYYMMDD) with the `DtpDate()` constructor.
2. Convert the SAP date string into the generic date format (YYYYMMDD HHMMSS) with the `getCWDate()` method.
3. Create the destination attribute and initialize its value to the generic date format with the `setWithCreate()` method.

The last section of the code should look like this:

```
// HERE IS THE MODIFIED CODE
if (_cw_SetSrcVal != null)
{
    try
    {
        DtpDate myDate = new DtpDate((String)_cw_SetSrcVal, "YMD");
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            myDate.getCWDate());
    }
    catch (DtpDateException de)
    {
        logError(5501);
        logInfo(de.getMessage());
    }
}
```

**Example:** To map the Clarify date attribute into the generic date, perform a Copy of the source attribute (the Clarify date string) into the destination attribute (the generic date string):

1. Create a `DtpDate` object to hold the generic date.  
Copy the source attribute (the Clarify date string) into a new `DtpDate` object (the generic date string) by parsing the Clarify date string (MM/DD/YYYY HH:MM:SS) with the `DtpDate()` constructor.
2. Convert the Clarify date string into the generic date format (YYYYMMDD HHMMSS) with the `getCWDate()` method.
3. Create the destination attribute and initialize its value to the generic date format with the `setWithCreate()` method.

The following code converts the Clarify date into the Generic date:

```
if (_cw_SetSrcVal != null)
{
    try
    {
        DtpDate myDate = new DtpDate((String)_cw_SetSrcVal,
            "M/D/Y h:m:s");
    }
}
```

```

        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            myDate.getCWDate());
    }
    catch (DtpDateException de)
    {
        logError(5501);
        logInfo(de.getMessage());
    }
}

```

To map the Generic date into the Clarify format, use the following code:

```

if (_cw_SetSrcVal != null)
{
    try
    {
        DtpDate myDate = new DtpDate((String)_cw_SetSrcVal, "YMD hms");
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            myDate.toString("M/D/Y h:m:s", true));
    }
    catch (DtpDateException de)
    {
        logError(5501);
        logInfo(de.getMessage());
    }
}

```

**Steps for converting from generic date format:** To convert from the generic date format to the SAP date format, perform the following steps:

1. Create a DtpDate object to hold the SAP date.  
Perform a Copy of the source attribute (the generic date string) into a new DtpDate object (the SAP date string) by parsing the generic date string (YYYYMMDD HHMMSS) with the DtpDate() constructor.
2. Convert the IBM generic date format into the SAP date string (YYYYMMDD) with the toString() method.
3. Create the destination attribute and initialize its value to the SAP date format with the setWithCreate() method.

The following code fragment shows this generic-to-SAP date conversion:

```

// HERE IS THE MODIFIED CODE
if (_cw_SetSrcVal != null)
{
    try
    {
        DtpDate myDate = new DtpDate((String)_cw_SetSrcVal, "YMD hms");
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            myDate.toString("YMD"));
    }
    catch (DtpDateException de)
    {
        logError(5501);
        logInfo(de.getMessage());
    }
}

```

### Obtaining the current date

To create a DtpDate object that is initialized with the current system date, use the DtpDate() constructor without any parameters:

```
DtpDate()
```

**Example:** To map the current date into the generic format, use the following code:

```
//get the current date
DtpDate myDate = new DtpDate();
// format the date to the destination object's format and map
_cw_SetDestBusObj.set(_cw_SetDestAttr, myDate.getCWDate());
```

To map the current date into the application-specific format, use the following code:

```
//get the current date
DtpDate myDate = new DtpDate();
// format the date to the destination object's format and map
_cw_SetDestBusObj.set(_cw_SetDestAttr, myDate.toString(
    "application format");
```

## Using Expression Builder for string transformations

When writing transformation code, you may need to build complex Java expressions to reference a particular attribute, manipulate a string, or call an API method. You can enter these expressions manually in Activity Editor, or use Expression Builder to construct the expression interactively. Expression Builder is a utility available from within Activity Editor.

**Tip:** Alternatively, you can use Activity Editor's Graphical view.

To display Expression Builder, place the cursor at the position in Activity Editor where you want to insert the expression and do one of the following:

- From the Tools menu, select Expression Builder.
- On the Java toolbar, click the Expression Builder button.
- Right-click anywhere in the Activity Editor window and select Expression Builder from the Context menu.

Figure 89 identifies the main components of Expression Builder.

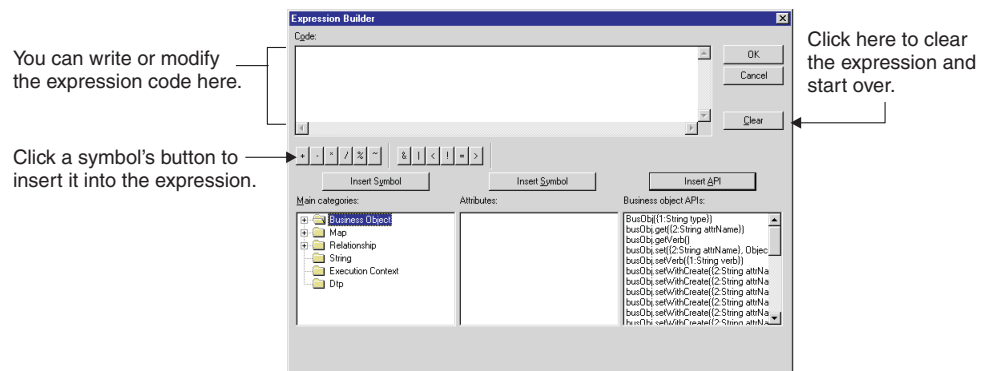


Figure 89. Expression Builder dialog

**Note:** In the API list, the number 1 refers to the object selected from the far left window and the number 2 refers to the object selected from the middle window, if there is an object present. When you insert the API, pairs of brackets surround variables (<<variable>>). You must replace the brackets and the variable with a value.

This section describes how to use Expression Builder to perform the following kinds of string transformations:

- “Steps for converting to uppercase text” on page 183
- “Using string manipulation methods” on page 183



## Steps for converting to uppercase text

Move `SAP_CustomerMaster.CustomerName` into `Customer.AccountOpenedBy` and convert all letters to uppercase text.

To use the `toUpperCase()` string manipulation functions, perform the following steps:

1. Perform a Move transformation from `SAP_CustomerMaster.CustomerName` into `Customer.AccountOpenedBy`.
2. Open the Activity Editor window by double-clicking in the Transformation Rule column associated with `Customer.AccountOpenedBy`. In Activity Editor, scroll down to the point where the destination is set to the source data. Change it to contain just the following code:

```
if (_cw_SetSrcVal != null)
{
    //
    // Since our version of simple data types are immutable in
    // Java (Strings included), we do not have to make a copy
    // of the source value here.
    //
    _cw_SetDestBusObj.setWithCreate(
        _cw_SetDestAttr, _cw_SetSrcVal);
}
```

The `_cw_SetSrcVal` variable contains `SAP.CustomerName` and `_cw_SetDestAttr` contains `AccountOpenedBy`.

3. Convert `CustomerName` to uppercase before copying it into `AccountOpenedB`. If you know the method to use, add it to the line of code above; otherwise, you can use Expression Builder to help you find the correct method. To do this:
  - a. Copy (Ctrl+C) `_cw_SetSrcVal`.
  - b. Highlight `_cw_SetSrcVal` (this code will be replaced with the code generated by Expression Builder) and select Expression Builder from the Tools menu.
  - c. Select String in the Main Categories list.
  - d. Select the method `toUpperCase()` from the list of String APIs.
  - e. Click Insert API.
  - f. When this method is displayed in the top window, replace the word "string," which is a placeholder, with the word `_cw_SetSrcVal` that you copied earlier.
  - g. Click OK.  
**Result:** The method call is inserted into the code.
  - h. Select the Save option from the File menu to save the code.

## Using string manipulation methods

The following string transformation uses string manipulation methods (such as `length()` and `substring()`) to move `SAP_CustomerMaster.AddressLine1` into `Customer.CustomerAddress.AddressLine1` and `AddressLine2`:

- If the length of `AddressLine1` is less than 10 characters, move the SAP `AddressLine1` to `CustomerAddress.AddressLine1` and do *not* map `AddressLine2`.
- If the length is greater or equal to 10 characters, map everything after the comma in the SAP `AddressLine1` into `CustomerAddress.AddressLine2`. If a comma is not found, map 9 characters to `AddressLine1`, and the rest to `AddressLine2`.

**Steps for performing the string transformation:** To perform this string transformation, follow these steps:

1. Perform a Move transformation of AddressLine1 into CustomerAddress.AddressLine1.
2. Open the Activity Editor window of AddressLine1 and scroll down to the point where the destination is set to the source data. Change it to contain the following code:

```
if (_cw_SetSrcVal != null)
    //
    // Since our version of simple data types are immutable in
    // Java (Strings included), we do not have to make a copy
    // of the source value here.
    //
    {
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            _cw_SetSrcVal);
    }
```

The `_cw_SetSrcVal` variable contains `SAP.AddressLine1`, and `_cw_SetDestAttr` contains `CustomerAddress.AddressLine1`.

3. Check the length of `SAP.AddressLine1` and either map the entire string into `CustomerAddress.AddressLine1` or map the substring of `SAP.AddressLine1` that precedes the comma.

You can either modify the code yourself or use Expression Builder. In either case, use the methods `length()`, `substring(int, int)`, `substring(int)`, and `indexOf(int)` methods of the `String` class.

4. Change the code to the following:

```
if (_cw_SetSrcVal != null)
    {
        // first check the length of _cw_SetSrcVal
        if (((String)_cw_SetSrcVal).length() < 10)
            {
                // if it is less than 10, map it to AddressLine1
                _cw_SetDestBusObj.setWithCreate(
                    _cw_SetDestAttr, _cw_SetSrcVal);
            }
        else
            {
                // if the length is not less than 10, search for comma
                int index = ((String)_cw_SetSrcVal).indexOf(",");

                // if comma is found, take a substring of _cw_SetSrcVal up to
                // the comma and map it to AddressLine1
                if (index != -1)
                    _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
                        ((String)_cw_SetSrcVal).substring(0, index));
                // if comma is not found, take first 9 characters of
                // _cw_SetSrcVal and map them to AddressLine1
                else
                    _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
                        ((String)_cw_SetSrcVal).substring(0, 9));
            }
    }
```

5. Move `SAP.AddressLine1` into `CustomerAddress.AddressLine2`. Modify the bottom part of the code to contain the following:

```
if (_cw_SetSrcVal != null)
    {
        // first check the length of _cw_SetSrcVal
        if (((String)_cw_SetSrcVal).length() >= 10)
            {
                // if the length is not less than 10, search for comma
                int index = ((String)_cw_SetSrcVal).indexOf(",");
                // if comma is found, take a substring of _cw_SetSrcVal after
                // the comma and map it to AddressLine2
```

```

    if (index != -1)
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            ((String)_Cw_SetSrcVal).substring(index + 1));
        // if comma is not found, take all characters of
        // _cw_SetSrcVal starting at the 10th and map them to
        // AddressLine1
    else
        _cw_SetDestBusObj.setWithCreate(_cw_SetDestAttr,
            ((String)_Cw_SetSrcVal).substring(9));
    }
}

```

### Source data validation

If you write code yourself, make sure that the source data is not null or blank. To check whether the source attribute is null, use one of the following:

- `if (ObjSource.isNull("Attr"))`— returns true if it is empty
- `if (ObjSource.get("Attr") == null)` — returns true if *attr* is empty

To check whether the source attribute is an empty string, do the following:

- `if (ObjSource.isBlank("Attr"))` — returns true if it is blank
- `if (ObjSource.getString("Attr").length() == 0)` — returns true if *attr* of type String is blank

---

## Reusing map instances

Typically, the map development system creates an instance of a map to process each transformation of data between the source and destination business objects. When the instance completes the handling of the transformation, the system frees up its resources. To reduce memory usage, the IBM system recycles an instance of a map instance by caching it and reusing it when the same type of map is instantiated at some later time. When the IBM system can recycle an existing map instance, it can avoid the overhead of map instantiation, thereby improving overall system performance and memory use.

**Restrictions:** The map development system automatically caches a map instance; that is, a map instance uses the Map Instance Reuse option by default. However, the Map Instance Reuse option imposes the following programming requirements on the map:

- Avoid using global variables in the map code.  
A global variable is a variable you declare in the Map local declaration block area of the General tab in the Map Properties dialog.
- If your map requires global variables, avoid initializing these global variables at declaration time. Instead, ensure that the global variables are always initialized at a map node, preferably the first transformation (attribute) node in a map.

**Attention:** A map containing global variables that are *not* initialized at the first transformation node cannot safely be recycled because the variable values in the cached map instance persist when the instance is reused. When the cached map instance is reused and begins execution, each global variable contains the value from the end of the previous use of the map instance.

If you cannot define your map so that it meets the preceding restrictions, you *must* disable the Map Instance Reuse option for this map. To disable this option, remove

the check mark from the Map instance reuse box, which appears in the map's Map Properties window in System Manager. This window also allows you to specify the size of the map-instance pool.

**Note:** Deploying the map to the server will not update the run-time instance. You can update the map properties dynamically from the server component management view by right-clicking on the map and selecting the properties from the Context menu. The changes will be automatically updated to the server.

---

## Handling exceptions

An *exception* represents an occurrence that, if not handled explicitly within the map, stops the map's execution. During the execution of a map, run-time exceptions can occur. When you define a custom transformation rule, you can use the "Catch Error" function block to trap any run-time exception. Once you catch a particular exception, you can determine how to handle this exception.

### Relationship exceptions

When using relationships in a map, several exceptions can occur. All of these exceptions are subclasses of `RelationshipRuntimeException`. If you are not concerned about the kind of exception, but simply want to catch them all, you can catch `RelationshipRuntimeException`. Otherwise, you can catch any of the following exceptions for specific cases:

- `RelationshipRuntimeDataAccessException`—thrown if a problem occurs while accessing the relationship database. You might catch this exception in any method call from the `Relationship` or `Participant` class.
- `RelationshipRuntimeDuplicateIdentityEntryException`—thrown if you try to add a participant to an identity relationship with the same relationship instance ID as an existing relationship instance. You might catch this exception in `addMyChildren()` and `create()` method calls.
- `RelationshipRuntimeUserErrorException`—is an abstract exception. It is thrown only if a `RelationshipRuntimeMetadataErrorException` or `RelationshipRuntimeGeneralUserErrorException` occurs. You might catch this exception in any method call from the `Relationship` or `Participant` class during map development. Once the map is debugged, you can remove the handlers for this exception.
- `RelationshipRuntimeMetadataErrorException`—thrown if an error occurs while manipulating the metadata associated with participant instances, such as the relationship name or participant definition name. You might catch this exception in any method call that adds, modifies, or deletes participant instances.
- `RelationshipRuntimeGeneralUserErrorException`—thrown if there is an error in the run-time data supplied with a `Relationship` or `Participant` class method call.

**Example:** The exception is thrown if you pass a business object of the wrong type to the `create()` method.

Figure 90 illustrates the relationship run-time exception hierarchy. Any exception you catch automatically catches those that are lower in the hierarchy. However, if an exception lower in the hierarchy is thrown, you cannot know exactly which one

it is unless you catch it specifically.

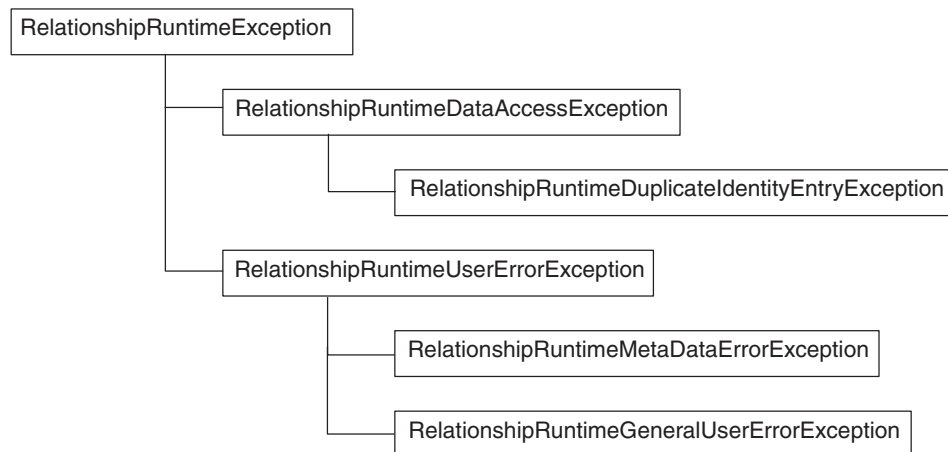


Figure 90. Relationship run-time exceptions

**Example:** If you catch `RelationshipRuntimeUserErrorException`, you automatically also catch `RelationshipRuntimeMetaDataErrorException` and `RelationshipRuntimeGeneralUserErrorException`. However, you cannot easily know which one of these was actually thrown, unless you test the exception with the instance of operator. The exception you choose to catch depends on how specific you want your exception handling to be.

## Example: Handling duplicate relationship instance IDs

When you add a participant to an identity relationship using the `addMyChildren()` or `create()` methods, `RelationshipRuntimeDuplicateIdentityEntryException` is thrown if you pass a relationship instance ID that already exists. This exception also provides a method, `getInstanceId()`, to retrieve the relationship instance ID that caused the duplication.

**Example:** The following example shows how you can catch this exception and retrieve the ID.

```
int instanceId;

try
{
    instanceId = Relationship.create(myParticipantObj);
}
catch (RelationshipRuntimeDuplicateIdentityEntryException rrdiee)
{
    /*
    ** There already is a relationship instance with this
    ** entry, grab the instanceId from the exception
    */
    instanceId = rrdiee.getInstanceId();
}
```

---

## Creating custom data validation levels

When values are mapped from one business object to another based on transformation code, incorrect data can result. The data validation feature checks each operation in a map and logs an error when data in the incoming business object cannot be transformed to data in the outgoing business object according to certain rules.

**Example:** Suppose that a map transforms a string value in the source business object to an integer value in the destination business object. This type conversion works properly when an incoming string value represents an integer (for example, "1234" represents the integer 1234). However, the conversion does not work properly if the string value does not represent an integer (for example, "ABCD" might indicate invalid data).

This section provides the following information about using data validation levels in a map:

- "Coding a data validation level"
- "Steps for testing the data validation level" on page 189

## Coding a data validation level

The map development system defines data validation levels 0 and 1; levels 2 and greater are available for you to define. Table 62 summarizes the data validation levels:

Table 62. Data Validation Levels

Level	Description
0	Default; no data validation
1	IBM-defined data type checks
2 and greater	User-defined validation checks

To create a custom validation level, double-click an attribute to display its code in the Activity Editor window. Note the following if statement, which introduces the code that throws an exception when invalid data is encountered:

```
if (dataValidationLevel >= 1)
```

Add your own if statement to specify the action that you want to associate with the data validation level. The data validation rules that you associate with a level can be whatever is appropriate and needed for your business logic and applications. For example, a level 2 rule for a particular attribute might check for a certain value and, if it is not present, set the attribute to a default.

**Example:** The following example code uses data validation level 2 to check that a credit card number is valid:

```
/*  
** At data validation level 2, check credit card numbers resulting  
** from data entry errors, etc.  
*/  
if (dataValidationLevel >= 2)  
{  
    if (!CustUtility.validateCreditCard(ObjPurchaseReq.getString  
        ("creditCardNumber")))  
    {  
        logWarning("Invalid credit card number sent through.");  
        if (failOnInvalidData)  
        {  
            throw new MapFailureException(  
                "Invalid credit card number.");  
        }  
    }  
}
```

## Steps for testing the data validation level

In a test run of the map, you can set the data validation level to see that it is working properly. Perform the following steps to test the data validation level for a map instance:

1. Display the Map Properties dialog, as follows:

From the Edit menu, select Map Properties. For information on other ways to display the Map Properties dialog, see “Specifying map property information” on page 58.

**Result:** The General tab of the Map Properties dialog appears.

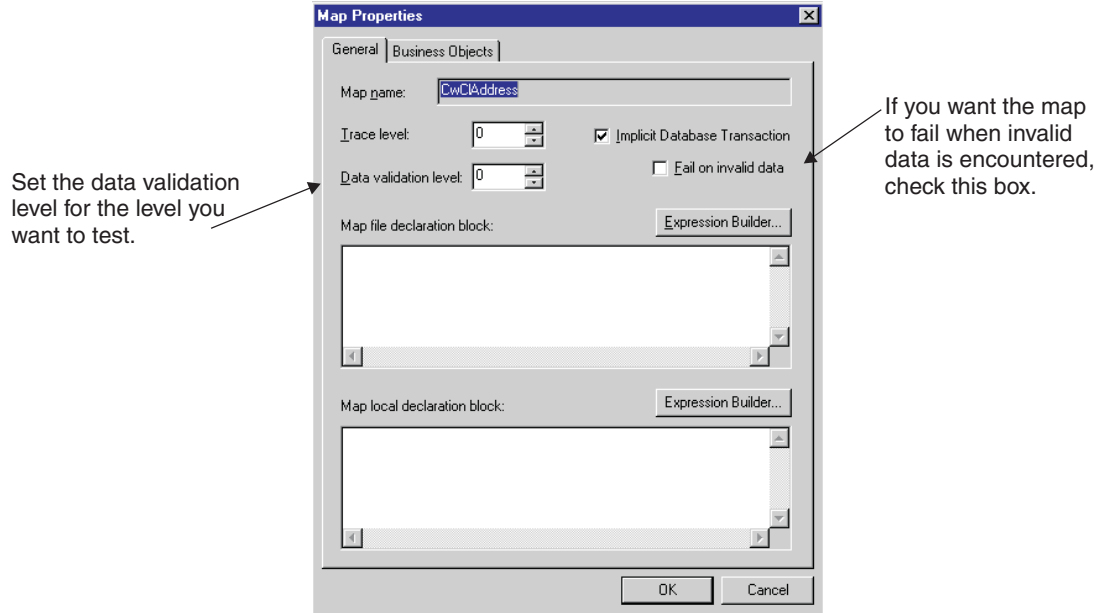


Figure 91. General tab of Map Properties dialog

2. Set the data validation level you want to test. Optionally, set the map to fail if the data is invalid.
3. Stop and restart the map for the new data validation level to take effect.
  - If you have changed the map code, recompile the map. Recompiling automatically restarts the map.
  - If you have *not* changed the map code, you do not have to recompile. However, you must explicitly stop and restart the map before the new data validation level takes effect. Use the Component menu of service Manager to stop and start a map.

**Note:** You can also set the data validation level and whether it fails if invalid data is present from the Map Properties window and the server component management view.

---

## Understanding map execution contexts

Each map instance executes within a specific *execution context* that is set by the connector controller. The Mapping API represents the map execution context with an instance of the MapExeContext class.

For every map that Map Designer generates, the map's execution context is accessible through a system-defined variable named `cxExecCtx`. You can reference this variable in the Variables folder in Activity Editor or when you call a Mapping API method that requires the execution context, including those in Table 63.

Table 63. Mapping API methods that require the map execution context

Purpose	Mapping API method	For more information
Calling a submap	<code>runMap()</code>	"Transforming with a submap" on page 43
Maintaining a relationship	<ul style="list-style-type: none"> <li>• <code>addMyChildren()</code></li> <li>• <code>deleteMyChildren()</code></li> <li>• <code>updateMyChildren()</code></li> <li>• <code>maintainChildVerb()</code></li> <li>• <code>maintainSimpleIdentityRelationship()</code></li> <li>• <code>maintainCompositeRelationship()</code></li> <li>• <code>foreignKeyXref()</code></li> <li>• <code>foreignKeyLookup()</code></li> </ul>	Chapter 8, "Implementing relationships," on page 257

Table 64 shows the two pieces of information that you most often need from the map execution context.

Table 64. MapExeContext Methods for Context Information

Execution Context Information	MapExeContext Method
The calling context	<code>getInitiator()</code> , <code>setInitiator()</code>
The original-request business object	<code>getOriginalRequestBO()</code>

This section discusses both these pieces of map-execution-context information.

**Note:** In addition, you can use the `getConnName()` and `setConnName()` methods of `MapExeContext` to access the name of the connector from the map execution context.

## Calling contexts

The *calling context* indicates the purpose for the current map execution. When transforming relationship attributes, you usually need to take actions based on the map's calling context. Table 65 lists the valid constants for calling contexts.

Table 65. Calling contexts

Calling-context constant	Description
<code>EVENT_DELIVERY</code>	The source business object(s) being mapped are event(s) from an application, sent from a connector to InterChange Server in response to a subscription request (event-triggered flow).
<code>ACCESS_REQUEST</code>	The source business object(s) being mapped are calls from an application, sent from an access client to InterChange Server (call-triggered flow).
<code>ACCESS_RESPONSE</code>	The source business object(s) being mapped are sent back to the access client in response to a subscription delivery request.
<code>SERVICE_CALL_REQUEST</code>	The source business object(s) being mapped are sent from InterChange Server to an application, through a connector.



Table 65. Calling contexts (continued)

Calling-context constant	Description
SERVICE_CALL_RESPONSE	The source business object(s) being mapped are sent back to InterChange Server from an application as a response to a successful service call request.
SERVICE_CALL_FAILURE	The source business object(s) being mapped are sent back to InterChange Server from an application after a failed service call request.

You can reference these calling contexts as constants in the MapExeContext object that is available in every map that Map Designer creates.

**Example:** You reference the SERVICE\_CALL\_REQUEST calling context as `MapExeContext.SERVICE_CALL_REQUEST`.

Figure 92 illustrates when each of the calling contexts occurs in an event-triggered flow. Event-triggered flow is initiated when a connector sends an event to a collaboration in InterChange Server.

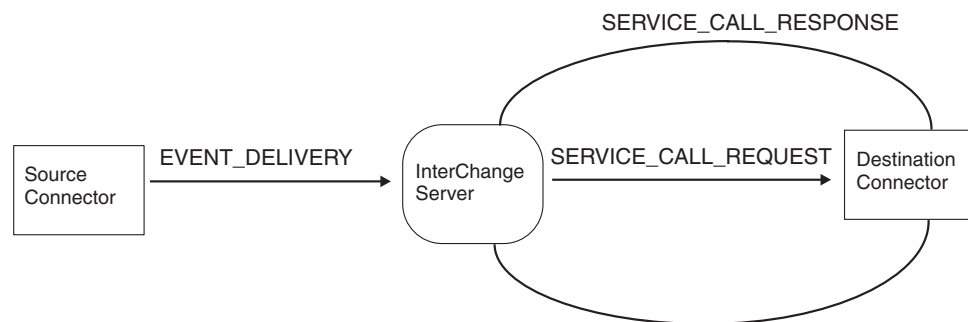


Figure 92. Calling contexts in an event-triggered flow

As Figure 92 shows, any mapping request coming from a connector to InterChange server (that is, a map from application-specific business object to generic business object) has a calling context of `EVENT_DELIVERY`. Any mapping request coming from InterChange server to a connector (that is, a map from generic business object to application-specific business object) has a calling context of `SERVICE_CALL_REQUEST`. Mapping requests sent by connectors in response to a collaboration's service call request can have contexts of `SERVICE_CALL_RESPONSE` or `SERVICE_CALL_FAILURE`.

Figure 93 illustrates when each of the calling contexts occurs in a call-triggered flow. Call-triggered flow is initiated when an access client sends a direct Server Access Interface call to a collaboration in InterChange Server.

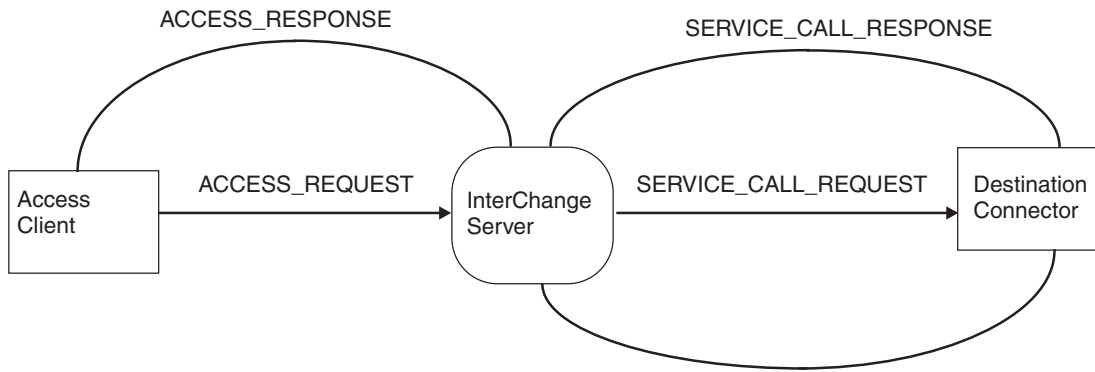


Figure 93. Calling contexts in a call-triggered flow

As Figure 93 shows, any mapping request coming from an access client to InterChange server (that is, a map from application-specific business object to generic business object) has a calling context of ACCESS\_REQUEST. Any mapping request coming from InterChange Server to an access client (that is, a map from generic business object to application-specific business object) has a calling context of ACCESS\_RESPONSE.

## Original-request business objects

Another important part of the map's context is the *original-request business object*. This business object is the one that has initiated the map execution. Table 66 shows the calling contexts and the associated original-request business object.

Table 66. Calling contexts and their associated original-request business objects

Calling context	Original-request business object	Original-request business object from example
EVENT_DELIVERY, ACCESS_REQUEST	Application-specific business object that came in from the application	AppA-specific
SERVICE_CALL_REQUEST, SERVICE_CALL_FAILURE	Generic business object that was sent down from InterChange Server	Generic
SERVICE_CALL_RESPONSE	Generic business object that was sent down by the SERVICE_CALL_REQUEST	Generic
ACCESS_RESPONSE	Application-specific business object that came in from the access request initially	AppA-specific

For example, the *generic business object* is the original-request business object for maps that execute with a calling context of SERVICE\_CALL\_RESPONSE, SERVICE\_CALL\_FAILURE, or SERVICE\_CALL\_REQUEST. These maps use the generic business object to store relationship instance IDs for the relationship attributes being transformed. Having the relationship instance IDs is necessary for the map to look up the relationship instance and fill in the relevant participant data for newly created or updated objects.

**Example:** The following example illustrates how this might work in a customer synchronization scenario. Suppose you are using the system to keep data synchronized between Application A and Application B. Both applications store customer data, and the customer ID attributes are managed using a relationship. For the purposes of this example, details about the collaborations and connectors involved are omitted.

When a new customer is added in Application A:

1. A map transforms an AppA-specific business object to a generic business object with a calling context of `EVENT_DELIVERY`.

When transforming the customer ID attribute, the map creates a new relationship instance in the customer ID relationship table and inserts the new relationship instance ID into the customer ID attribute of the generic business object.

2. A map transforms the generic business object to a AppB-specific business object with a calling context of `SERVICE_CALL_REQUEST`.

No changes occur to the relationship tables. Application B successfully adds the new customer to the application.

3. A map transforms the AppB-specific business object to a generic business object with a calling context of `SERVICE_CALL_RESPONSE`. The context for this map execution includes the generic business object generated in step 1.

The reason for this execution is to fill in the new participant data for the relationship instance created in step 1. In this case, the new participant data is the customer ID for the new customer added to application B.

Figure 94 illustrates when the map execution for each step occurs for a call-triggered flow that successfully adds a new customer ID to Application B.

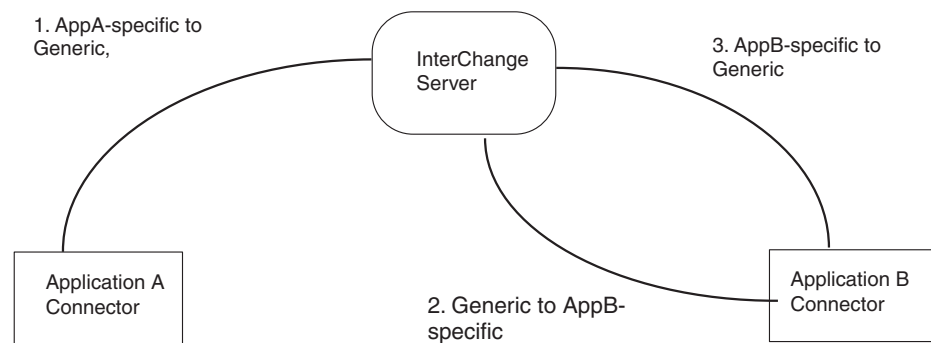


Figure 94. Example of Calling Contexts

## Mapping child business objects

When the source business object contains child business objects, how you map the child business object depends on the cardinality of the child in the source and destination business objects. This section provides information on how to map child business objects in the following cases:

- “Mapping single-cardinality source and destination”
- “Mapping single-cardinality source to multiple-cardinality destination” on page 194
- “Mapping multiple-cardinality source and destination” on page 194

### Mapping single-cardinality source and destination

To map a single-cardinality source child business object to a single-cardinality destination business object:

- Click the plus symbol (+) to the left of the source child object name to expand it, and follow simple transformation steps (see “Specifying standard attribute

transformations” on page 36 and “More attribute transformation methods” on page 173) to map source attributes to the destination attributes. You can create a submap for it, but it is not necessary.

- It is a good practice to set a verb for the destination child object. To do this, follow the instructions in “Setting the destination business object verb” on page 36. Set the verb to the source parent object’s verb (unless the objects have special relationship requirements).

**Example:** This kind of mapping would be needed for the following map:

SAP\_CustomerMaster

to

Customer.CustomerAddress

## Mapping single-cardinality source to multiple-cardinality destination

If the source child business object is of cardinality 1, it contains data to be mapped only to a single instance of the destination child business object. To map a single-cardinality source child object to a multiple-cardinality destination child object, follow the procedure described in “Mapping single-cardinality source and destination” on page 193. There is no need to create a submap.

**Example:** This kind of mapping would be needed for the following map:

SAP\_CustomerMaster.SAP\_CustCreditCentralData[1]

to

Customer.CustomerInformation.CustomerCreditData[n]

## Mapping multiple-cardinality source and destination

To map a multiple-cardinality source child business object to a multiple-cardinality destination business object, you need to create a multiple-cardinality submap. For an introduction to submaps, see “Transforming with a submap” on page 43.

**Example:** This kind of mapping would be needed for the following map:

SAP\_CustBankData[n]

to

Customer.CustomerInformation.CustomerBankData[n]

When you transform multiple source business objects of the same type, perform the subtasks outlined in Table 67.

*Table 67. Creating a multiple-cardinality submap*

Subtask	Associated procedure (see . . . )
1. Creating a multiple-cardinality submap, which performs the transformations for attributes in one source object to one destination business object.	“Steps for creating the multiple-cardinality submaps” on page 195
2. Calling this submap from the main map, in the multiple-cardinality attribute of the destination business object. The call to the submap is within a for loop that loops through each business object within the multiple-cardinality object so that each business object is passed to the submap.	“Steps for calling the multiple-cardinality submap” on page 195

## Steps for creating the multiple-cardinality submaps

To create a submap that maps multiple-cardinality child objects from the source to destination object, you create a map with a single source business object and the single destination business object. This map contains the transformations for attributes in the source object to the corresponding attributes in the destination object.

To create a multiple-cardinality submap, perform the following steps from within Map Designer:

1. Close your main map and start a new map.
2. In the Diagram tab, drag the source child business object into the left half of the map workspace area and the destination child business object into the right half.

To map the SAP\_CustBankData business object to the Customer.CustomerInformation.CustomerBankData business object, drag SAP\_CustBankData into the left half of the workspace and CustomerBankData into the right half.

3. Save the submap.

**Recommendation:** You should begin submap names with the prefix “Sub\_”. For example: Sub\_SaCwCustBankData.

4. Set the verb of the destination object as described in “Setting the destination business object verb” on page 36.
5. Map individual attributes as described in “Specifying standard attribute transformations” on page 36 and “More attribute transformation methods” on page 173.
6. Compile the submap by selecting Compile from the File menu.

**Result:** If everything is correct, the following message displays:

```
Map validation OK.  
Map compilation successful.
```

**Tip:** If you forget to compile the submap, you cannot see it in the Submap dialog. The message No submaps available displays.

## Steps for calling the multiple-cardinality submap

To call the multiple-cardinality submap, call it with the runMap() method from the main map, in the multiple-cardinality attribute of the destination business object. The call to the submap is within a for loop that loops through each business object within the multiple-cardinality object so that each business object is passed to the submap.

To call a multiple-cardinality submap, perform the following steps from within Map Designer:

1. Close the submap and open the main map.
2. Select the source child object and the destination child object and select Submap from the transformation rule’s combo box.

For the SAP\_CustBankData to Customer.CustomerInformation.CustomerBankData transformation, select SAP\_CustBankData and CustomerBankData.

**Result:** The Submap dialog box displays.

3. Select the name of the submap and click OK.

**Example:** Sub\_SaCwCustBankData

In this example, you do not need to specify a condition, so click OK.

4. Open the Activity Editor of the destination child object.

For the SAP\_CustBankData to Customer.CustomerInformation.CustomerBankData transformation, you see code similar to the following in Activity Editor:

```

{
BusObjArray srcCollection_For_ObjSAP_Order_SAP_OrderPartners =
    ObjSAP_Order.getBusObjArray("SAP_OrderPartners");

//
// LOOP ONLY ON NON-EMPTY ARRAYS
// -----
//
// Perform the loop only if the source array is non-empty.
//
if ((srcCollection_For_ObjSAP_Order_SAP_OrderPartners != null) &&
    (srcCollection_For_ObjSAP_Order_SAP_OrderPartners.size() > 0))
    {
    int currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners;
    int lastInputIndex_For_ObjSAP_Order_SAP_OrderPartners =
        srcCollection_For_ObjSAP_Order_SAP_OrderPartners.getLastIndex();
    for (currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners = 0;
        currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners <=
            lastInputIndex_For_ObjSAP_Order_SAP_OrderPartners;
        currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners++)
        {
        BusObj currentBusObj_For_ObjSAP_Order_SAP_OrderPartners =
            (BusObj) (srcCollection_For_ObjSAP_Order_SAP_OrderPartners.elementAt(
                currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners));

//
// INVOKE MAP ON VALID OBJECTS
// -----
//
// Invoke the map only on those child objects that meet certain
// criteria.
//
if (currentBusObj_For_ObjSAP_Order_SAP_OrderPartners != null)
    {
    BusObj[] _cw_inObjs =
        { currentBusObj_For_ObjSAP_Order_SAP_OrderPartners };
    BusObj[] _cw_outObjs =
        DtpMapService.runMap(
            "Sub_SaOrderPartners_to_CwCustomerRole", "CwMap",
            _cw_inObjs, cwExecCtx);
    ObjOrder.setWithCreate("AssociatedCustomers",
        _cw_outObjs[0]);
    }
    }
}
}

```

Notice that runMap() is a static method, so it is invoked as:

```
DtpMapService.runMap()
```

5. Make a few changes to this code, as follows:

- In the source business object, only the parent object has a verb associated with it. In the code for calling the submaps to map the child objects, it is good practice to set the source child object's verb to the one of the parent's before passing this child object into the submap. Use the setVerb() method of the BusObj class.

**Note:** If relationship management (cross-referencing) has to be performed in the submap, verb passing *is required*. See "Defining transformation rules for a simple identity relationship" on page 273 for more information.

- The line of code for calling the submap must be included inside of the try/catch block to ensure that you catch the MapNotFoundException.

Here is the modified code (changes are highlighted in bold):

```

    {
        BusObjArray srcCollection_For_ObjSAP_
Order_SAP_OrderPartners =      ObjSAP_Order.getBusObjArray("SAP_OrderPartners");

        //
        // LOOP ONLY ON NON-EMPTY ARRAYS
        // -----
        //
        // Perform the loop only if the source array is non-empty.
        //
        if ((srcCollection_For_ObjSAP_Order_SAP_OrderPartners
            != null) &&
            (srcCollection_For_ObjSAP_Order_SAP_OrderPartners.size()
            > 0))
        {
            int currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners;
            int lastInputIndex_For_ObjSAP_Order_SAP_OrderPartners =
                srcCollection_For_ObjSAP_Order_SAP_OrderPartners.getLastIndex();

            for (currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners
                = 0;
                currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners
                <=
                lastInputIndex_For_ObjSAP_Order_SAP_OrderPartners;
                currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners++)
            {
                BusObj currentBusObj_For_ObjSAP_Order_SAP_OrderPartners =
                (BusObj)
                    (srcCollection_For_ObjSAP_Order_SAP_OrderPartners.elementAt(
                    currentBusObjIndex_For_ObjSAP_Order_SAP_OrderPartners));

                //
                // INVOKE MAP ON VALID OBJECTS
                // -----
                //
                // Invoke the map only on those child objects that meet
                // certain
                // criteria.
                //
                if (currentBusObj_For_ObjSAP_Order_SAP_OrderPartners != null)
                {
                    currentBusObj_For_ObjSAP_Order_SAP_OrderPartners.setVerb(
                    ObjSAP_Order.getVerb());          BusObj[] _cw_inObjs

                    = { currentBusObj_For_ObjSAP_Order_SAP_OrderPartners };

                    try
                    {
                        BusObj[] _cw_outObjs = DtpMapService.runMap(
                            "Sub_SaOrderPartners_to_CwCustomerRole", "CwMap",
                            _cw_inObjs, cwExecCtx);
                        ObjOrder.setWithCreate("AssociatedCustomers", _cw_outObjs[0]);
                    }
                    catch (MapNotFoundException me)
                    {
                        logError(5502, "Sub_SaOrderPartners_to_CwCustomerRole");
                        throw new MapFailureException ("Submap not found");
                    }
                }
            }
        }
    }
}

```

6. Once you have added the call to the submap, recompile the main map.

---

## More on using submaps

This section provides the following tips on the use of submaps:

- “Providing conditions when calling the submap”
- “Using Expression Builder to call a submap” on page 200
- “Passing business objects of different types to submaps” on page 201

For an introduction to submaps, see “Transforming with a submap” on page 43.

### Providing conditions when calling the submap

Often the map requires some programming logic to determine when to call a submap. Sometimes you might have certain conditions that must be true for the submap to be called. This logic goes into the main map, in the attribute that contains the submap’s destination business object, and before the call to the `runMap()` method. If you enter these conditions in the Conditions area of the Submap dialog (see Figure 21 on page 46), Map Designer adds these conditions to the `if` statement with which it surrounds the `runMap()` call.

**Guidelines:** Keep the following points in mind when entering these conditions:

- You can enter only one condition, although the condition can have multiple clauses, combined with an AND (&&) operator.
- Do not end the line with a semicolon because the condition that you enter is turned into an `if` clause in the destination attribute’s generated code.

**Example:** In the business objects shown in Figure 20 on page 44 the `OrderLine` business object has an attribute called `DelSched`, which is a child business object. In a submap condition, you can refer to that attribute as follows:

```
srcBusObj
```

Enter the following submap condition in the Conditions area of the Submap dialog to execute a submap on the `DelSched` business object *only if* the value of the `TransportType` attribute of the same business object is equal to the string `AIR`.

```
srcBusObj.getString("TransportType").equals("AIR")
```

The following condition executes the submap for `OrderLine DelSched` attributes *only if* the `OrderLine LinePrice` attribute value is greater than \$10,000.00.

```
ObjOrderLine.getFloat("LinePrice") > 10000.00
```

**Example:** Conditions are needed for the following map because the mapping should occur *only if* `SAP_CustPartnerFunctions.PartnerId` is *not equal to* `SAP_CustomerMaster.CustomerId`:

```
SAP_CustomerMaster.SAP_CustSalesAreaData.SAP_CustPartnerFunctions[n]
```

to

```
Customer.RelatedCustomerRef
```

The following sections take you through the steps in creating the map call:

- “Steps for creating the submap” on page 199
- “Steps for calling the submap” on page 199



## Steps for creating the submap

To create the Sub\_SaCwCustCreditAreaData submap, perform the following steps from within Map Designer:

1. Close the main map and start a new submap.  
Specify a source business object of SAP\_CustPartnerFunctions and a destination business object of RelatedCustomerRef. Name the map as Sub\_SaCwCustPartners.
2. Set the verb as described in “Setting the destination business object verb” on page 36.
3. Map individual attributes as described in “Specifying standard attribute transformations” on page 36 and “More attribute transformation methods” on page 173.
4. Compile the submap by selecting Compile from the File menu. If everything is correct, the following message displays:  
Map validation OK.  
Native Map: Code generation succeeded.

## Steps for calling the submap

To call the Sub\_SaCwCustCreditAreaData submap, perform the following steps from within Map Designer:

1. Go back to the main map and, in the Diagram tab, hold the Ctrl key and drag SAP\_CustPartnerFunctions onto RelatedCustomerRef. Double-click Submap from the list in the Rule column of the destination business object. Select the SaCwCustPartners map in the Submap dialog and click OK.

In the case of a simple condition, you can type it into the Condition area for Submap dialog window:

```
srcBusObj.getString("PartnerId").equals("some_id")
```

Map Designer generates the code for the submap call with the condition added to the if statement that precedes the submap call.

If you leave the Condition field blank and click OK, Map Designer generates the code for the submap call without the condition. Continue to step 2 for information on how to explicitly add the code.

2. Modify the generated code to add the submap condition.

To access Activity Editor, double-click the Rule column again and click the View Code push-button of the Submap dialog.

For the RelatedCustomerRef attribute, the code that includes the condition follows:

```
{
    BusObjArray srcBusObjs = null;
    srcBusObjs =
ObjSAP_CustomerMaster.getBusObjArray
("SAP_CustSalesAreaData.SAP_CustPartnerFunctions");
    String parent_custid = ObjSAP_CustomerMaster.getString("CustomerId");
    //
    // LOOP ONLY ON NON-EMPTY ARRAYS
    // -----
    //
    // Perform the loop only if the source array is non-empty.
    //
    if ((srcBusObjs != null) && (srcBusObjs.size() > 0))
    {
        int srcIndex;
        int lastSrcIndex = srcBusObjs.getLastIndex();
        for (srcIndex = 0; srcIndex <= lastSrcIndex; srcIndex++)
        {
            BusObj srcBusObjInstance = (BusObj)
```

```

(srcBusObjs.elementAt(srcIndex));
    //
    // INVOKE MAP ON VALID OBJECTS
    // -----
    //
    // Invoke the map only on those children objects
    // that meet certain criteria.
    //
    if ((srcBusObjInstance != null))
    {
        // check the submap running condition
        if
        (
            (!(srcBusObjInstance.getString("PartnerId")).equals(parent_custid))
            {
                // set the verb of the source business object of submap
                srcBusObjInstance.setVerb(ObjSAP_CustomerMaster.getVerb());
                try
                {
                    BusObj[] _cw_inObjs = { srcBusObjInstance };
                    BusObj[] _cw_outObjs =
                DtpMapService.runMap("Sub_SaCwCustPartners",
                "CwMap", _cw_inObjs,cwExecCtx);
                ObjCustomer.setWithCreate("RelatedCustomerRef", _cw_outObjs[0]);
                }
                catch(MapNotFoundException me)
                {
                    logError(5502, "Sub_SaCwCustPartners");
                    throw new MapFailureException("Submap not found");
                }
            }
        }
    }
}
}
}
}
}
}
}
}
}
}

```

3. Once you have added the call to the submap, recompile the main map.

## Using Expression Builder to call a submap

You can use Map Designer to automatically generate a submap call through the Submap dialog. However, you can also use Expression Builder to generate the submap call. Coding the map call allows you to write more varied operations than Map Designer supports graphically. For example, you can use a submap to provide a value for an attribute that does not contain a child business object or use a submap that has multiple inputs and outputs.

### Steps for using Expression Builder to call a submap

Perform the following steps to use Expression Builder to generate a submap call:

1. Double-click the transformation rule in either the Table or Diagram tab to display Activity Editor in Java view.
2. Right-click anywhere in Activity Editor, then select Expression Builder from the Context menu.
3. In Expression Builder:
  - In the Main categories column, expand the Map folder.
  - Under Maps, expand the map type: Native Map folder to display a list of all compiled maps that are available on your system.
  - Select the map to call and select the `DtpMapService.runMap()` method in the Map APIs column.
  - Click the Insert API button to initiate code generation (see Figure 95).

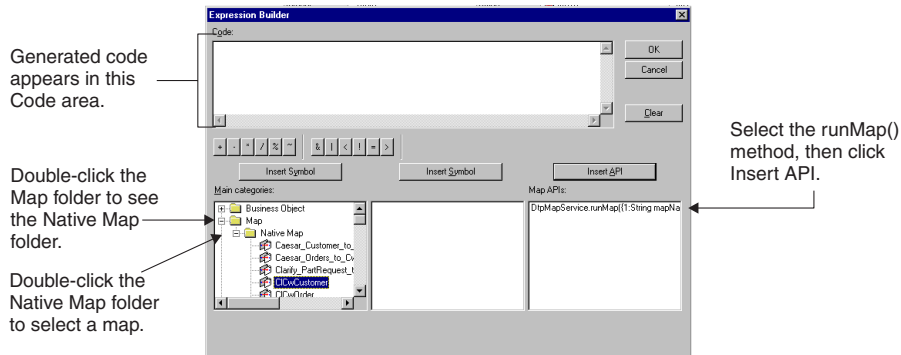


Figure 95. Coding a submap call

**Result:** Map Designer generates the Java code necessary to call the map you specify as a submap. It displays in the code area at the top of the Expression Builder window.

4. In the generated code, replace the input and output object place holders with the actual current source and destination business object names for the submap, and write code to handle the value that the submap returns.

Replace the placeholders `InObj`, `OutObj`, and `OutAttrName` with the correct names for the following:

- The top-level source business object variable: `ObjSrcBusObj`
- The source attribute: the path to the name of the attribute that contains the child business object
- The destination business object variable: `ObjDestBusObj`
- The destination attribute.

5. Click OK to close Expression Builder.

**Result:** The code is inserted into Activity Editor at the insertion point.

## Passing business objects of different types to submaps

To map source business objects of different types to a destination business object, you need to create a many-to-one submap.

**Example:** This kind of mapping would be needed for the following map:

```
SAP_CustCreditControlAreaData[n]
```

```
to
Customer.CustomerInformation.CustomerCreditData[0].
CreditAreaCreditData[n]
```

Some of the attributes must be mapped from `SAP_CustomerMaster` and `SAP_CustCreditCentralData`.

Transforming multiple source business objects of different types involves the following major steps:

1. “Steps for creating the many-to-one submap” on page 202—create a many-to-one submap, which transforms the needed attributes in the multiple source business objects to those in the single destination business object.
2. “Steps for calling the many-to-one submap” on page 202—Call this submap from the main map, in the attribute that holds the submap’s destination business object.

## Steps for creating the many-to-one submap

To create a submap that maps source child business objects of different types to one destination child object, you create a map with the many source business objects and the single destination business object. This map contains the transformations for attributes in the different source object to the corresponding attributes in the destination object.

To create a many-to-one submap, perform the following steps from within Map Designer:

1. Close the main map and start a new map. The new map will have at least two source objects and one destination object.

Drag SAP\_CustCreditControlAreaData, SAP\_CustCreditCentralData, and SAP\_CustomerMaster business objects into the left half of the mapping screen, and CreditAreaCreditData into the right half. The new map will have three source objects and one destination object.

2. Save the submap.

**Recommendation:** Submap names should begin with the prefix "Sub\_". For example: Sub\_SaCwCustCreditAreaData.

3. Set the verb of the destination object as described in "Setting the destination business object verb" on page 36.

Unless the relationship management is performed in this submap, it does *not* matter which source object is used to set the verb. Ensure that when you call this submap, you set the verb for the same particular object before passing it into the submap. If one of the source objects is the source parent object and you choose to use its verb, you do not need to set the verb before passing this object into the submap; it already has a verb associated with it.

4. Map individual attributes as described in "Specifying standard attribute transformations" on page 36 and "More attribute transformation methods" on page 173.

5. Compile the submap by selecting Compile from the File menu.

**Result:** If everything is correct, the following message displays:

```
Map validation OK.  
Native Map: Code generation  
succeeded.
```

## Steps for calling the many-to-one submap

To invoke the many-to-one submap, call it with the runMap() method from the main map, in the attribute that holds the submap's destination business object. To invoke the Sub\_SaCwCustCreditAreaData submap, perform the following steps from within Map Designer:

1. Open the main map.

For the Sub\_SaCwCustCreditAreaData submap, the main map is the map that contains the CreditAreaCreditData business object as an attribute.

2. Double-click in the Transformation Rule column for the submap's destination object to open Activity Editor.

Double-click the Transformation Rule column associated with the CreditAreaCreditData attribute to open its Activity Editor and enter the following code:

```
{  
  BusObj srcobj1 = ObjSAP_CustomerMaster;  
  BusObj srcobj2 = (BusObj)  
    (ObjSAP_CustomerMaster.getBusObj("SAP_CustCreditCentralData"));  
  BusObjArray srcobj3 =
```

```

        (BusObjArray)(ObjSAP_CustomerMaster.get(
            "SAP_CustCreditControlAreaData"));

//
// INVOKE MAP ON VALID OBJECTS
// -----
//
// Invoke the map only on those child objects that meet certain
// criteria.
//
int i = 0;

// When checking all 3 source objects, != null might be not required
if (srcobj1 != null && srcobj2 != null & srcobj3 != null)
{
    for (i = 0; i < srcobj3.size(); i++)
    {
        BusObj[] _cw_inObjs = new BusObj[3];

        // set verb for the one of the following objects
        _cw_inObjs[0] = srcobj1;
        _cw_inObjs[1] = srcobj2;
        _cw_inObjs[2] = srcobj3.elementAt(i);

        try
        {
            BusObj[] _cw_outObjs = DtpMapService.runMap(
                "Sub_SaCwCustCreditAreaData",
                "CwMap",
                _cw_inObjs,
                cwExecCtx);
            ObjCustomer.setWithCreate(
                "CustomerInformation.CustomerCreditData[0].CreditAreaCreditData["
                + i + "]", _cw_outObjs[0]);
        }

        catch (MapNotFoundException me)
        {
            {
                logError(5502, "Sub_SaCwCustCreditAreaData");
                throw new MapFailureException ("Submap not found");
            }
        }
    }
}

```

3. To track the map execution, use the `logInfo()` method inside the for loop. The code follows.

```
logInfo("in for loop");
```

or

```
trace("in for loop");
```

4. Once you have added the call to the submap, recompile the main map.

If everything is working properly and none of your source objects is null, the message in for loop displays in the InterChange Server log file as many times as there are instances of the source child object.

---

## Executing database queries

During execution of a map, you might need to obtain information from a database, such as the relationship database. To obtain or modify information in 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 68 shows the steps involved in executing a query in a database.

**Note:** You can access any external database that IBM supports through JDBC through the Oracle thin driver and the WebLogic driver. Databases include Oracle and Microsoft SQL Server

Table 68. Subtasks for executing a query

Subtasks for executing a query	Associated procedure (see . . . )
1. Obtaining a connection (which is a <code>CwDBConnection</code> object) to the database.	“Obtaining a connection” on page 204
2. Through the <code>CwDBConnection</code> object, sending queries and managing transactions in the database.	“Executing the query” on page 204 “Managing the transaction” on page 216
3. Releasing the connection.	“Releasing a connection” on page 219

## 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 `BaseDLM` 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 map instance that requests the connection. You configure connection pools within IBM System Manager.

In Figure 96, 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 96. 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 216.

## Executing the query

Table 69 shows the ways that you can execute SQL queries with methods of the `CwDBConnection` class.

Table 69. 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>

Table 69. Executing SQL queries with CwDBCConnection methods (continued)

Type of query	Description	CwDBCConnection method
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.	executePreparedStatement()
Stored procedure	A user-defined procedure that contains SQL statements and conditional logic	executeSQL()executePreparedStatement() executeStoredProcedure()

## 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.

**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 `CwDBCConnection` object obtained from a previous call to the `getDBCConnection()` method (see Figure 96).

You can also send a SELECT statement that has parameters in it by using the second form of the `executeSQL()` method.

**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, perform these steps:

1. Obtain one row of data.
2. Obtain column values, one by one.

Table 70 shows the methods in the `CwDBConnection` class that provide access to the rows of returned query data.

*Table 70. 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 71 shows the Java methods for accessing the columns of a returned query row.

*Table 71. Java Methods for column-value access*

Column-access task	Java method
Determine number of columns.	<code>Vector.size()</code>
Cast <code>Vector</code> to <code>Enumeration</code> .	<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.

**Example:** 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());
}
}

```

**Example:** 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 = theRow.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.

**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 CwDBCConnection object obtained from a previous call to the getDBCConnection() 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, you should assess the need for transaction

management for these statements. For more information, see “Managing the transaction” on page 216.

**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 211.

### Executing prepared queries

The `executePreparedSQL()` 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 97.
- When you execute a SELECT statement with `executePreparedSQL()`, you still use the same methods to access the returned data (Table 70 and Table 71). For more information, see “Executing static queries that return data (SELECT)” on page 205.

**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 re-execute 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 provide information dynamically to each statement execution.

The code fragment in Figure 97 inserts 50 rows into the employee table. The first time `executePreparedStatement()` 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), `executePreparedStatement()` 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.executePreparedStatement(
            "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 97. 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 re-execute 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 216.

**Note:** To simplify the code in Figure 97 does *not* include transaction management.

**Executing a prepared stored procedure:** You can use the `executePreparedStatement()` 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 211.

## 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 72 shows the methods in the `CwDBConnection` class that call a stored procedure.

Table 72. *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>	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 should use 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 from 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 205.

As Table 72 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 `CwDBCConnection`:

- 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()` 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 98, the call to `executeSQL()` sends a CALL statement to execute the `setOrderCurrDate()` stored procedure.

```
connection.executeSQL("call setOrderCurrDate(345698)");
```

*Figure 98. Calling a stored procedure with executeSQL()*

In Figure 98, the `connection` variable is a `CwDBCConnection` object obtained from a previous call to the `getDBCConnection()` 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 212.

**Note:** Use an anonymous PL/SQL block if you plan on calling Oracle stored PL/SQL objects via ODBC using the `CwDBCConnection.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.

*Steps for calling a stored procedure with the `executeStoredProcedure()` method:* To call a stored procedure with the `executeStoredProcedure()` method, perform the following steps:

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 73 shows the parameter information that a `CwDBStoredProcedureParam` object contains as well as the methods to retrieve and set this parameter information.

*Table 73. Parameter information in a `CwDBStoredProcedureParam` object*

Parameter information	<code>CwDBStoredProcedureParam</code> method
Parameter value	<code>getValue()</code>
Parameter in/out type	<code>getParamType()</code>

*Steps for passing parameters to a stored procedure with `executeStoredProcedure()`:* To pass parameters to a stored procedure with `executeStoredProcedure()`, perform the following steps:

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.

**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 99 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 99. 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.

**Example:** A JDBC INTEGER is normally mapped to a Java `int` type. The `executeStoredProcedure()` method uses the mappings shown in Table 74.

Table 74. Mappings between Java and JDBC data types

Java data type	JDBC data type
String	CHAR, VARCHAR, or LONGVARCHAR
Integer, int	INTEGER
Long	BIGINT
Float, float	REAL
Double, double	DOUBLE
<code>java.math.BigDecimal</code>	NUMERIC

Table 74. Mappings between Java and JDBC data types (continued)

Java data type	JDBC data type
Boolean, boolean	BIT
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.sql.Clob	CLOB
java.sql.Blob	BLOB
byte[]	BINARY, VARBINARY, or LONGVARBINARY
Array	ARRAY
Struct	STRUCT

## 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 217

### 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 is determined programmatically.

At run time, a map instance determines which transaction programming model to use for each connection it acquires. By default, a map 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 75.

Table 75. Overriding a transaction programming model for a connection

Transaction programming model to override	Action to take
To specify a transaction programming model <i>for all connections</i> obtained by a particular map instance	Select or deselect the Implicit Database Transaction check box on the General tab of the Map Properties dialog. You can also set this property on the Map Properties window of Service Manager.
To specify a transaction programming model <i>for a particular connection</i>	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.  The following <code>getDBConnection()</code> call specifies explicit transaction bracketing for the connection obtained from the ConnPool connection pool: <pre>conn = getDBConnection("ConnPool", false);</pre>

You can determine the current transaction programming model that connections will use with the `BaseDLM.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"

**Transaction scope with implicit transaction bracketing:** InterChange Server handles the actions of the map in a single implicit transaction. If the connection uses implicit transaction bracketing, InterChange server also handles the transaction management for operations performed on an external database, one associated with a connection from a connection pool. When a map performs database operations, InterChange Server also handles these database operations as an implicit transaction, which is a subtransaction of the main transaction (the map). This database subtransaction begins as soon as the map obtains the connection. InterChange Server implicitly ends this subtransaction when execution of the map completes.

The success or failure of this database subtransaction depends on the success or failure of the map, as follows:

- If the map is successful, InterChange Server commits the database subtransaction.
- If the map fails, InterChange Server rolls back the database subtransaction. If this rollback fails, InterChange Server throws the `CwDBTransactionException` exception and logs an error.

**Transaction scope with explicit transaction bracketing:** If the connection uses explicit transaction bracketing, InterChange Server expects the map definition 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 map.

**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 map.

Table 76 shows the methods in the `CwDBConnection` class that provide management of transaction boundaries for explicit transactions.

*Table 76. CwDBConnection methods for explicit transaction management*

Transaction-management task	CwDBConnection method
Begin a new transaction.	<code>beginTransaction()</code>
End the transaction, committing (saving) all changes made during the transaction to the database.	<code>commit()</code>
Determine if a transaction is currently active.	<code>inTransaction()</code>
End the transaction, rolling back (backing out) all changes made during the transaction.	<code>rollback()</code>

*Steps for specifying the transaction scope of an explicit transaction:* To specify transaction scope of an explicit transaction, perform the following 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* the 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 map. If the map is successful, InterChange Server commits this database transaction. If the map is *not* successful, InterChange Server implicitly rolls back the database transaction. Regardless of the success of the map, InterChange Server 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.executeUpdate("update table1....");
    connection.executeUpdate("update table2....");
    connection.executeUpdate("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

InterChange Server automatically releases a connection that uses implicit transaction bracketing once it has ended the database transaction. InterChange Server does not end the database transaction until it determines the success or failure of the map; that is, InterChange Server releases these connections when the map instance finishes execution. If the map executes successfully, InterChange Server automatically commits any database transactions that are still active. If the map execution fails (for instance, if an exception is thrown that is not handled with a catch statement), InterChange Server 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:

- InterChange Server automatically releases a connection that uses explicit transaction bracketing.
- You can 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, InterChange Server does not end the database transaction until it determines the success or failure of the map. 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. InterChange Server automatically releases a connection regardless of the transaction programming model it uses. In most cases, you do not need to explicitly release the connection.

## Re-establishing a connection

Database connections can be broken due to various reasons such as a network problem or database shutdown. The InterChange Server can detect broken connections in both internal and user databases.

If a connection is broken, the InterChange Server automatically tries to re-establish the connect according to a predefined number of retry attempts at predefined time intervals. See *System Administration Guide* for more information on setting maximum retry attempts and retry time intervals.

**Note:** The default for maximum retry attempts is 3 and retry time intervals is 60 seconds.

---

## Part 2. Relationships





---

## Chapter 6. Introduction to relationships

This chapter provides an overview of WebSphere business integration relationships and the relationship development process.

This chapter covers the following topics:

- “What is a relationship?” on page 223
- “Relationships: A closer look” on page 229
- “Overview of the relationship development process” on page 235

---

### What is a relationship?

When attributes in a source and destination business object contain equivalent data that is represented differently, the transformation step employs a *relationship*. A relationship establishes an association between data from two or more business objects. Each business object is called a *participant* in the relationship.

The data that you typically transform using relationships are:

- ID numbers, such as a customer ID or product ID
- Other values represented as codes, such as country, currency, or marital status

**Example:** Suppose application A uses sequential integers for customer IDs, and application B uses generated customer codes. TashiCo has a customer ID of 806 in application A and A100 in application B. To transfer customer ID data between applications A and B, you can create a relationship among the application A customer business object, the generic customer business object, and the application B customer business object, based on the customer ID attributes.

This relationship establishes an association between customers from application A and application B, based on the key attributes of their customer business objects. In Figure 100, each box represents a participant in a relationship called CustIden.

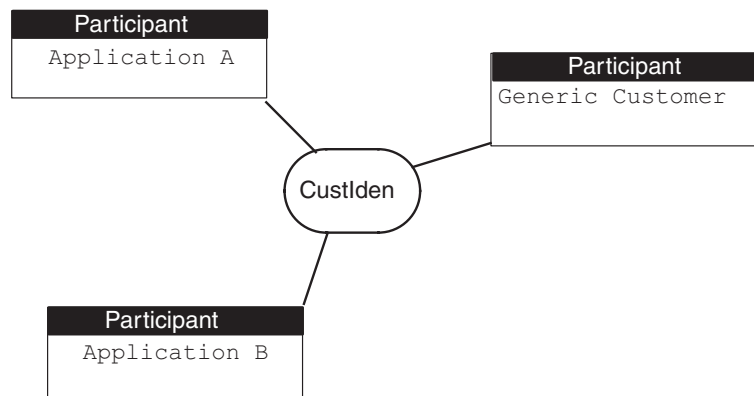


Figure 100. Relationship with three participants

Relationships are classified into the following categories based on the type of data in the participant and the number of instances of each participant that can be related:

- A *lookup relationship* establishes an association between data, such as attributes in business objects. The data can be related on a *one-to-one*, *one-to-many*, or *many-to-many* basis. Lookup relationships typically transform non-key attributes whose values are represented with codes, such as marital status or currency code. Use a lookup relationship if these attribute values are static; that is, new values are not often added or existing values removed.
- An *identity relationship* establishes an association between business objects or other data on a *one-to-one* basis. For each relationship instance, there can be only one instance of each participant. Identity relationships typically transform the key attributes of business objects, such as ID numbers and product codes. The relationship in Figure 100 is an example of an identity relationship. Use an identity relationship if key values are dynamic; that is, key values are frequently added or existing values are removed.
- A *non-identity relationship* establishes an association between business objects or other data on a *one-to-many* or *many-to-many* basis. For each relationship instance, there can be one or more instances of each participant. An example of a non-identity relationship is an RMA-to-Order transformation, in which a single RMA (Return Materials Authorization) business object can yield one or more Order business objects.

## Lookup relationships

A *lookup relationship* relates two pieces of non-key data. For example, in a Clarify\_Site to Customer map, you might transform attributes whose values are represented by codes or abbreviations, such as SiteStatus, using a lookup relationship. In a lookup relationship, there is one participant for each application-specific business object.

The CustLkUp relationship in Figure 101 establishes a lookup relationship between customer status codes from Clarify and SAP applications. Each box represents a participant in the CustLkUp lookup relationship. Notice that this relationship has two participants, one for each application-specific business object.

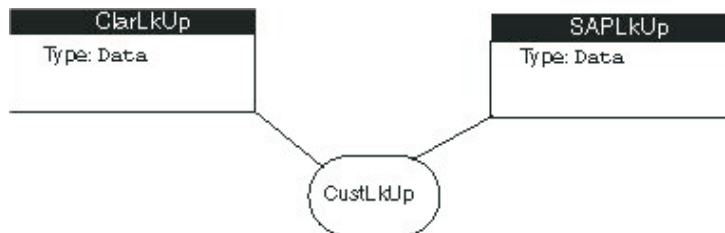


Figure 101. CustLkUp lookup relationship definition

**Note:** Because a lookup relationship does not indicate which attributes are being related, its participants use a special type called Data. For more information, see “Participant type” on page 233.

**Example:** Suppose that the Clarify application represents an inactive customer with a site status of Inactive while in SAP the corresponding value is 05. Although these customer status codes are different, they represent the same status, as Figure 102 shows.

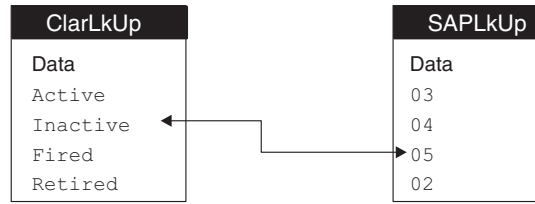


Figure 102. Relationship data for the CustLkUp lookup relationship

Table 77 shows the steps needed to create a lookup relationship.

Table 77. Steps for creating a lookup relationship

Creation step	For more information
1. Define a lookup relationship in Relationship Designer.	"Defining lookup relationships" on page 246
2. Customize mapping code to maintain the lookup relationship.	"Using lookup relationships" on page 258
3. Test the lookup relationship to verify that it is implemented correctly.	"Testing a lookup relationship" on page 98

## Identity relationships

An *identity relationship* establishes an association between business objects or other data on a *one-to-one* basis. To maintain a one-to-one relationship, each business object must have a key; that is, the object contains at least one attribute (a *key attribute*) whose value uniquely identifies the object. If both business objects contain a key, they can participate in an identity relationship.

The WebSphere business integration system supports the following kinds of identity relationships:

- "Simple identity relationships"
- "Composite identity relationships" on page 227

Both kinds of identity relationships involve relating business object attributes. Therefore, each participant in an identity relationship has a business object as its participant type. For more information on participant types, see "Participant type" on page 233.

### Simple identity relationships

A *simple identity relationship* relates two business objects through a single key attribute; that is, each business object contains a single value that uniquely identifies the object.

**Example:** Suppose the CustIden relationship (see Figure 100) is further refined to establish an association between customers from the Clarify and SAP applications, based on the key attributes of their customer business objects. In Figure 103, each box represents a participant in this customer identity relationship. Notice that this relationship has a participant for each application-specific business object and the generic business object.

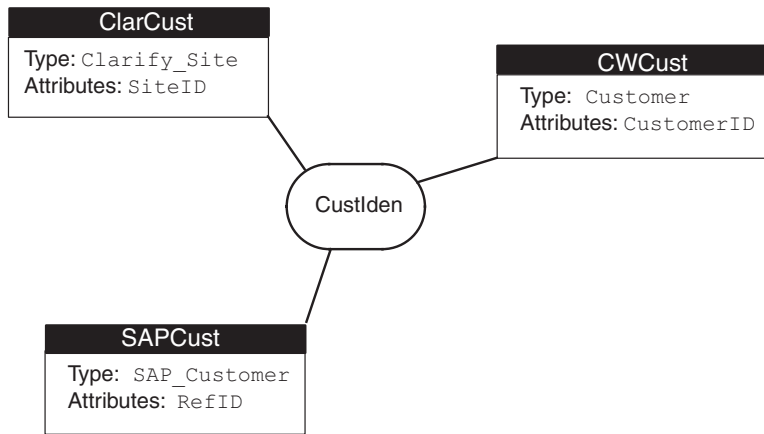


Figure 103. CustIden simple identity relationship definition

The TashiCo company is identified with a key value of A100 in the Clarify application while this same company is identified with a key value of 806 in the SAP application. Although these application IDs are different, they represent the same customer, as Figure 104 shows.

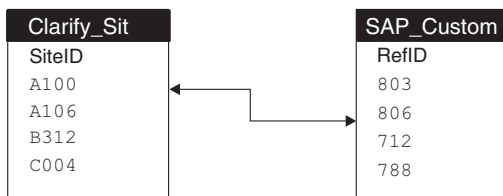


Figure 104. Relationship data for the custIden simple identity relationship

Therefore, the following maps use a simple identity relationship to maintain the transformations between the key attributes:

- The inbound maps (between the Clarify application-specific business object and the generic Customer business object) use a simple identity relationship to maintain the transformation between the SiteID attribute of the Clarify\_Site business object and generic CustomerID attribute of the generic Customer business object.
- The outbound maps (between the generic Customer business object and the SAP application-specific business object) also use a simple identity relationship to maintain the transformation between the RefID attribute of the SAP\_Customer business object and the generic CustomerID attribute of the generic Customer object.

Table 78 shows the steps needed to create a simple identity relationship.

Table 78. Steps for creating a simple identity relationship

Creation step	For more information
1. Define a simple identity relationship in Relationship Designer.	“Defining identity relationships” on page 244
2. Customize mapping code to maintain the simple identity relationship.	“Using simple identity relationships” on page 263
3. Test the simple identity relationship to verify that it is implemented correctly.	“Testing an identity relationship” on page 95

## Composite identity relationships

A *composite identity relationship* relates two business objects through a composite key. As the term “composite” indicates, a composite key is a key that consists of several attributes. Values for *all* attributes are needed to uniquely identify the object. A composite key consists of a unique key from a parent business object and a nonunique key from a child business object.

**Example:** Suppose a particular order from TashiCo in the Clarify application is identified with a key value of 8765. This same order in the SAP application is identified with a key value of 0003411. Because these two order numbers uniquely identify the same order, their key attributes are related with a simple identity relationship. However, an order also contains order lines. If all participating applications identify these order lines with a unique value, a simple identity relationship can maintain their transformations.

However, it is often the case that an application uses only the line number to identify an order-line item. That is, each order contains a line item identified with 1, with any subsequent items numbered 2, 3, and so on. These line numbers do *not* uniquely identify the order-line items. To uniquely identify such items, the application uses a composite key that consists of the order number (from the parent order business object) and the line number (from the child order-line business object).

In Figure 105, the `OrderLine` relationship establishes a relationship between order lines from the Clarify and SAP applications, based on their composite key attributes: the unique key attribute of their parent order business object combined with the order-line number in their child order-line business object. Each box represents a participant in the `OrderLine` composite identity relationship. Notice that each participant has two attributes.

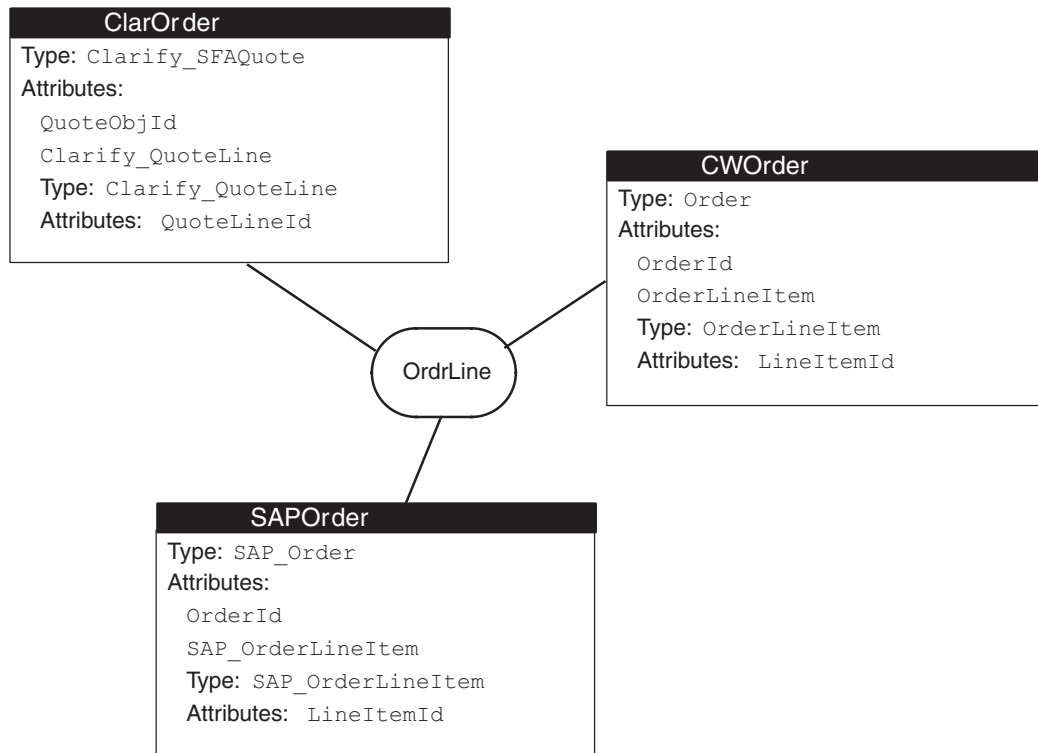


Figure 105. OrdrLine composite identity relationship definition

**Example:** Suppose the Clarify application (represented by the participant ClarOrder in Figure 105) uses sequential integers to identify order-line items, while the SAP application uses the line number to identify these items. The Clarify application uniquely identifies each order-line item. Therefore, the maps between the Clarify application-specific business object and the generic Order business object (represented by the participant CWOrder) can use a simple identity relationship to maintain the transformation of the order-line items.

However, the SAP application (represented by the participant SAPOrder) identifies order-line items with their line number. Its items are not uniquely identified: every order contains a line item identified with 1, with any subsequent items numbered 2, 3, and so on. To uniquely identify the third order-line item of Order 0003411, you need to use a composite key, which includes both the order number (0003411) and the item number (3). Therefore, the maps between the SAP application-specific business object and the generic Order business object must use a composite identity relationship to maintain the transformation of the order-line items.

The third line item from the TashiCo order (8765) is identified in the Clarify application with the simple key value of 1171. However, this same line item is identified in the SAP application with a composite key value of 0003411 (order number) and 3 (line number). Although these order lines are identified differently, they represent the same order line item, as Figure 106 shows.

Clarify_SF AQuote		SAP_Order	
QuoteObjId	Clarify_QuoteLine	OrderId	SAP_OrderLineItem
8764		0003409	
8765	1168	0003410	1
8765	1169	0003410	1
8765	1170	0003411	2
8766	1171	0003411	1
8766	1172	0003411	2
	1173		3

Figure 106. Relationship data for the OrdLine composite identity relationship

Table 79 shows the steps needed to create a composite identity relationship.

Table 79. Steps for creating a composite identity relationship

Creation step	For more information
1. Define a composite identity relationship in Relationship Designer.	“Defining identity relationships” on page 244
2. Customize mapping code to maintain the composite identity relationship.	“Using composite identity relationships” on page 274
3. Test the composite identity relationship to verify that it is implemented correctly.	“Testing an identity relationship” on page 95

## Relationships: A closer look

To understand the types of relationships that the WebSphere business integration system supports, you must understand how IBM implements the following concepts:

- “Relationships”
- “Participants” on page 233

## Relationships

As Table 80 shows, a relationship is a two-part entity, consisting of a repository entity and a run-time object.

Table 80. Parts of a relationship

Repository entity	Run-time object
Relationship definition	Relationship instance

### Relationship definition

You define a relationship to the WebSphere business integration system with a *relationship definition*. Relationship definitions identify each participant and specify how the participants are related. In Figure 100, CustIden is the relationship definition and it includes information about the three participants, Application A, Application B, and Generic Customer.

The system stores relationship definitions in the repository. The Relationship Designer tool provides dialogs to help you create the relationship definitions. Using this tool, you also store the completed relationship definition in the repository.

**Tip:** For more information on how to use Relationship Designer to create relationship definitions, see “Customizing the main window” on page 240.

The relationship definition provides the following information about the relationship:

- The relationship name
- The name of the relationship database

**Relationship definition name:** A relationship definition is simply a template, or description, of the relationship; it is *not* an actual business object. Therefore, the name of the relationship definition should *not* be the name of the associated business object.

**Relationship database:** The *relationship database* holds the relationship tables for a relationship. The relationship uses these relationship tables to keep track of the related application-specific values. For more information, see “Relationship tables” on page 231.

To access the relationship database at run time, the system must have the following information:

- The type of database management system (DBMS) that manages the relationship database
- The name and password of the user account that accesses the relationship database
- The location of the relationship database

By default, the relationship database is the WebSphere business integration system repository; that is, Relationship Designer creates all relationship tables in the repository. Relationship Designer allows you to specify the location of relationship tables in either of the following ways:

- Change the default location of relationship databases of *every* relationship.  
For more information, see “Global default settings” on page 251..
- Customize the location of each relationship’s tables as part of the process of creating a relationship definition.  
For more information, see “Advanced settings for relationship definitions” on page 249.

### Relationship instance

The relationship definition is a template for the run-time instantiation of the relationship, called the *relationship instance*. During map execution, the system creates instances of the relationship based on the relationship definition and using the values from the actual business objects being transformed.

**Example:** The relationship data for the CustLkUp lookup relationship (see Figure 102) shows that a customer status of Inactive in a Clarify application is the same as a customer status of 05 in an SAP application. Although these status codes are different, they represent the same customer status and therefore are in the same relationship instance, as Figure 107 shows.



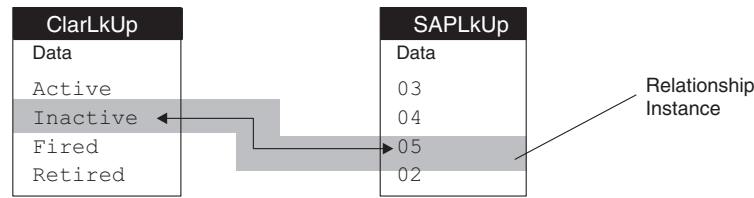


Figure 107. One Relationship instance for the CustLkUp relationship

A relationship instance is represented in the Mapping API by an instance of the Relationship or IdentityRelationship class.

To locate a relationship instance, the system requires the following information:

- A *relationship table* to identify which table contains the relationship instances for a particular participant
- A *relationship instance ID* to identify the actual relationship instance within the relationship table

**Relationship tables:** A *relationship table* is a database table that holds the relationship run-time data for one participant in a relationship. InterChange Server stores relationship instances in relationship tables, with one table (sometimes called a *participant table*) storing information for one participant in the relationship. For example, for the CustLkUp lookup relationship in Figure 101, InterChange Server requires two participant tables, as shown in Figure 107.

When you create a relationship definition, Relationship Designer automatically creates the table schemas that the relationship requires; that is, it creates the relationship tables with the necessary columns for each participant. At run time, these tables hold the data for the relationship instances.

**Note:** For an identity relationship, InterChange Server automatically populates the relationship tables. For a lookup relationship, you must populate the relationship tables with data. For more information, see “Populating lookup tables with data” on page 259.

To access a relationship table at run time, the system must have the following information:

- The name of the relationship table

Because a relationship table is associated with a participant, the name of this table is defined as part of the participant definition. By default, any relationship table has a name of the form:

*RelationshipDefName\_ParticipantDefName*

Relationship Designer allows you to customize the name of a relationship table as part of the process of creating a participant definition.

For more information, see “Advanced settings for participant definitions” on page 250.

- The name of the database that contains the relationship table

The name of the relationship database is set as part of the relationship definition. By default, the relationship database is the system repository. For more information, see “Advanced settings for relationship definitions” on page 249..

In map-transformation steps, relationship tables are managed using methods in the Relationship, IdentityRelationship, and Participant classes. Some Mapping API

methods automatically manage relationship tables. You can also explicitly access these relationship tables to obtain this relationship data.

**Relationship instance ID:** The WebSphere business integration system uniquely identifies each relationship instance by assigning it a unique integer value, called a *relationship instance ID*. This instance ID allows the system to correlate the participant values. In general, given any participant in a relationship, you can retrieve the data for any other participant in the relationship by specifying the relationship instance ID.

**Example:** For the relationship between customer status codes of a Clarify application and an SAP application, the WebSphere business integration system assigns a relationship instance ID to each relationship instance of the lookup relationship. Figure 108 shows how instance ID 47 associates the two application-specific participants, ClarLkUp and SAPLkUp. ID 47 associates the Clarify customer status of Inactive with the SAP customer status value of 05. Notice that this relationship is basically the same as the one in Figure 107, with the addition of the relationship instance ID.

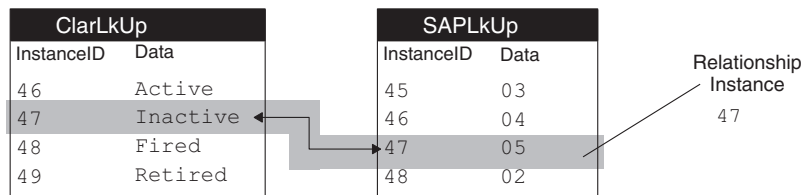


Figure 108. A lookup relationship with relationship instance IDs

The WebSphere business integration system also uses a relationship instance ID for the relationship between participants in an identity relationship. In the CustIden relationship (see Figure 103), this instance ID associates the customer IDs stored in the SiteID attribute of the Clarify\_Site business object, the CustomerID attribute of the generic Customer business object, and the RefID attribute of the SAP\_Customer business object. Figure 109 shows how the relationship instance data for each participant of the CustIden relationship is associated using the relationship instance ID.

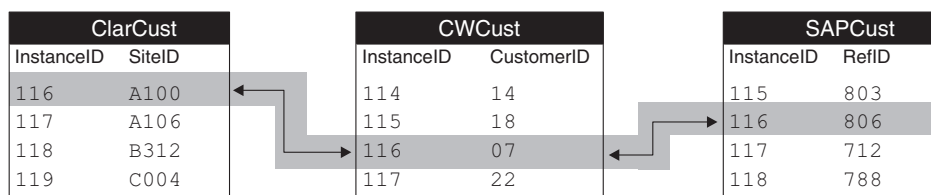


Figure 109. A customer identity relationship with relationship instance IDs

In Figure 109, the relationship table for the CWCust participant is included for clarity, though the table is not strictly necessary. In fact, relationship tables for the participant representing the generic business object in any relationship are necessary *only* if you want to generate a generic ID for the associated attribute in the generic business object. The relationship in Figure 109 generates a generic ID (07) for the CustomerID attribute in the generic Customer business object.

You can simplify your relationship definition and increase performance by eliminating the relationship tables for the participant that represents the generic business object. You do this by selecting the managed option for the participant

when you create the relationship definition. See “Advanced settings for participant definitions” on page 250 for more information about this setting.

Figure 110 shows how relationship instance data is associated in the CustIden relationship when the managed setting is specified for the CWCust participant.

ClarCust	
InstanceID	SiteID
116	A100
117	A106
118	B312
119	C004

SAPCust	
InstanceID	RefID
115	803
116	806
117	712
118	788

Figure 110. An identity relationship Instance with no generic table

The WebSphere business integration system stores the relationship instance ID in the relationship table for each participant. As Figure 108 through Figure 110 show, each relationship table in a relationship has a column that contains the relationship instance ID. InterChange Server automatically creates the instance ID column when it creates the table schema.

## Participants

A relationship contains *participants*, which describe the entities participating in the relationship. As Table 81 shows, a participant is a two-part entity, consisting of a repository definition and a run-time object.

Table 81. Parts of a participant

Repository entity	Run-time object
Participant definition	Participant instance

### Participant definitions

The relationship definition contains a list of *participant definitions*. For instance, the CustIden relationship definition in Figure 103 associates customer business objects in Clarify and SAP and contains these participant definitions: SAPCust, CWCust, and ClarCust.

The WebSphere business integration system stores participant definitions in the repository. The Relationship Designer tool provides dialogs to help you create the participant definitions. Using this tool, you also store the completed participant definition in the repository.

The participant definition provides the following information about the participant:

- The participant name
- The participant type
- The name of the participant table and stored procedures

**Participant definition name:** A participant definition is simply a template, or description, of the participant; it is *not* an actual business object. Therefore, the name of the participant definition should *not* be the name of the associated business object.

**Participant type:** Like the attributes in a business object definition, the participants in a relationship definition have an associated type. The participant

type specifies the kind of data associated with instances of the participant. The participant type can be one of the following:

- The name of a business object definition  
Relationships with participants of this type establish an association between entire business objects. In this case, you specify the attributes of the business object that relate the participant to the other participants in the relationship. The attributes you select, usually the key attributes of the business object, become the *participant instance identifiers*.
- The word Data  
In the participant definition, Data represents a supported attribute data type, such as String, long, int, double, float, or boolean. You specify Data as the type for participants in relationships that establish associations between specific attributes in business objects. Participants in lookup relationships have a participant type of Data.

For information on how to define the type of a participant, see “Creating a relationship definition” on page 243.

**Participant table and stored procedures:** For every participant, InterChange Server creates the following database entities:

- A participant table to hold the relationship instance IDs and the associated participant’s application-specific value
- Stored procedures to perform Retrieve (Select), Insert, Delete, and Update operations on the participant table

By default, Relationship Designer assigns names of the following form to the participant’s table and stored procedure: *RelName\_ParticipantName\_X*, where *RelName* is the name of the relationship definition, *ParticipantName* is the name of the participant definition, and *X* is T for the participant table or SP for the stored procedure. By default, Relationship Designer creates the relationship tables in the WebSphere business integration system repository.

Relationship Designer allows you to customize the names of the participant table and stored procedures. For more information on naming the participant table and stored procedures, see “Advanced settings for participant definitions” on page 250.

## Participant instances

The participant definition is a template for the run-time instantiation of the participant, called the *participant instance*. During map execution, the WebSphere business integration system creates instances of the participant based on the participant definition and the attribute values from the actual business objects being transformed.

The WebSphere business integration system stores participant instances as a column in the participant’s relationship table.

**Example:** For the CustIden relationship in Figure 103, the Clarcust participant has a column called SiteID in its participant table to hold the values of its participant instances. The SAPCust participant has a RefID column in its participant table to hold the values of its participant instances.

Each participant instance contains the following information:

- Name of the relationship definition
- Relationship instance ID

- Name of the participant definition
- Data to associate with the participant

A participant instance is represented in the Mapping API by an instance of the `Participant` class.

---

## Overview of the relationship development process

A relationship in the WebSphere business integration system is a two-part entity:

- A relationship definition, stored in the repository, to define the participants
- Code within a map to implement the relationship by accessing the relationship tables

To define a relationship in the WebSphere business integration system, you must perform the following basic steps:

1. Determine the type of relationship you need.
2. Within Relationship Designer define a relationship definition and define the composite participants.
3. Within Map Designer customize the transformation rule, if necessary, to maintain the relationship.
4. Recompile the affected maps.
5. Deploy the relationships and maps to InterChange Server with the Create Schema option.
6. Ensure that the relationship database(s) exists and is defined correctly within the relationship definition.
7. Populate relationship tables for any lookup relationships. Optionally, populate other relationship tables with test data for the testing phase.
8. For each map, start all relationships in the map.
9. Test the relationship with the Test Connector. Be sure to set the appropriate calling context as part of each of the tests.

Figure 111 provides a visual overview of the relationship development process and provides a quick reference to chapters where you can find information on specific topics. Note that if a team of people is available for map development, the major tasks of developing a map can be done in parallel by different members of the development team.

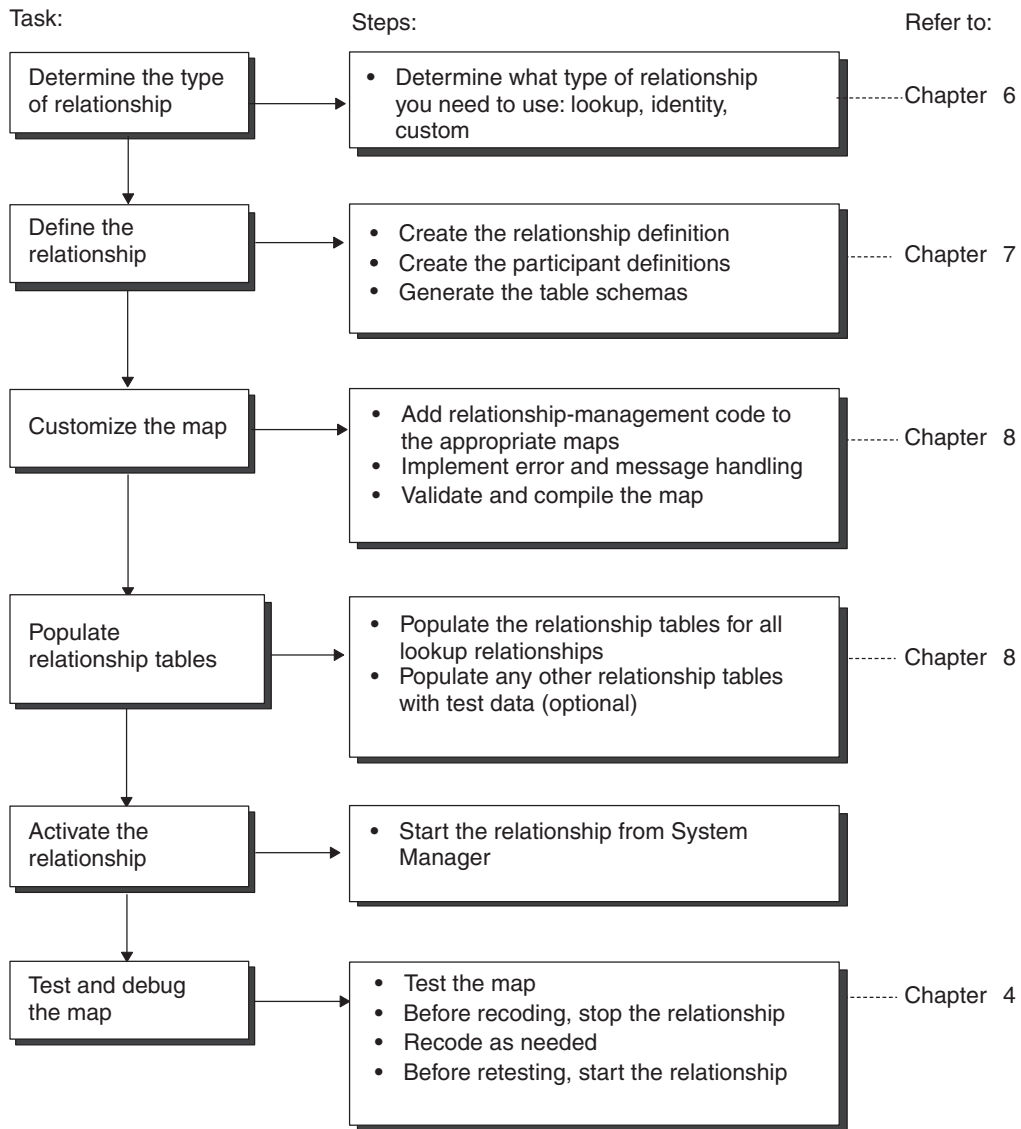


Figure 111. Overview of the relationship development task

---

## Chapter 7. Creating relationship definitions

This chapter describes how to create and modify relationship definitions using Relationship Designer. For background information on how the WebSphere business integration system uses relationships in mapping, see Chapter 6, “Introduction to relationships,” on page 223. For help customizing relationships in maps, see Chapter 5, “Customizing a map,” on page 103.

This chapter covers the following topics:

- “Overview of Relationship Designer” on page 237
- “Creating a relationship definition” on page 243
- “Defining identity relationships” on page 244
- “Defining lookup relationships” on page 246
- “Creating the relationship table schema” on page 248
- “Copying relationship and participant definitions” on page 248
- “Renaming relationship or participant definitions” on page 249
- “Specifying advanced relationship settings” on page 249
- “Deleting a relationship definition” on page 253
- “Optimizing a relationship” on page 253

---

### Overview of Relationship Designer

Relationship Designer is a graphical development tool for creating and modifying relationship definitions. A *relationship definition* establishes an association between two or more participants. You create a relationship definition by specifying the participants in the relationship and defining the data source and other properties associated with each participant.

This section covers the following topics to introduce you to Relationship Designer:

- “Starting Relationship Designer” on page 237
- “Working with projects” on page 238
- “Layout of Relationship Designer” on page 239
- “Customizing the main window” on page 240
- “Using Relationship Designer functionality” on page 241

### Starting Relationship Designer

To launch Relationship Designer, do one of the following:

- From System Manager, perform one of these actions:
  - From the Tools menu, select Relationship Designer.
  - Click a Relationship folder in a project to enable the Relationship Designer icon in the System Manager toolbar. Then click the Relationship Designer icon.
  - Right-click the Relationships folder in a project and select Relationship Designer from the Context menu.
  - Right-click a relationship in the Dynamic or Static folder and select Edit Definitions from the Context menu.

**Result:** Relationship Designer launches and highlights the selected relationship.

- From a development tool, such as Business Object Designer, Map Designer, perform one of these actions:
  - From the Tools menu, select Relationship Designer.
  - In the Programs toolbar, click the Relationship Designer button.
- Use a system shortcut:  
Start > Programs > IBM WebSphere InterChange Server  
> Toolset > Development > Relationship Designer

**Important:** For Relationship Designer to be able to access relationships stored in System Manager, Relationship Designer must be connected to an instance of System Manager. The preceding steps assume that you have already started System Manager. If System Manager is already running, Relationship Designer will automatically connect to it.

## Working with projects

System Manager is the only tool that interacts with the server. It imports and exports entities (relationships, maps) between InterChange Server and System Manager projects. Various tools, such as Relationship Designer, connect to System Manager and view, edit, and modify these entities on a project basis.

A *project* is simply a logical grouping of entities for managing and deployment purposes. Once entities are deployed to InterChange Server, the project they originated from no longer has any meaning.

System Manager allows you to create multiple projects. Before you can work on a relationship, you must select which project the relationship is in.

### Steps for selecting a project

To select a project to work with, perform the following steps:

1. From the File menu, select Switch to Project.
2. In the Switch to Project submenu, select the name of the project.

**Result:** You can now work with the relationships in that project. Before you can switch to yet another project, you need to save the relationships you modified in the current project.



Figure 112 shows the Switch to Project option for browsing a project.

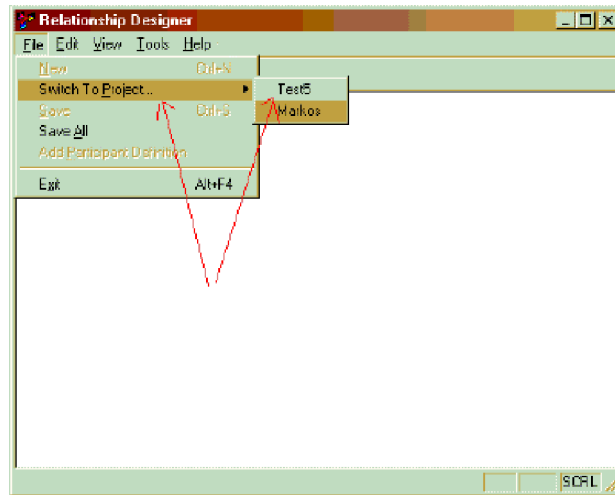


Figure 112. Browsing a project

When Relationship Designer establishes a connection to System Manager, it obtains a list of business objects that are defined in the current project. This list assists you with defining participants.

If you add or delete a business object using Business Object Designer, System Manager notifies Relationship Designer, which dynamically updates the list of business object definitions.

## Layout of Relationship Designer

In the Relationship Designer window, a list of relationship definitions stored in the current project appears on the left side. In this relationship definition list, the contents of each relationship definition appear in a hierarchical format similar to the Windows Explorer. You can expand the relationship name by clicking on the plus symbol (+) beside its name to see a list of its participant definitions, participant types, and associated attributes. Figure 113 shows a relationship definition list.

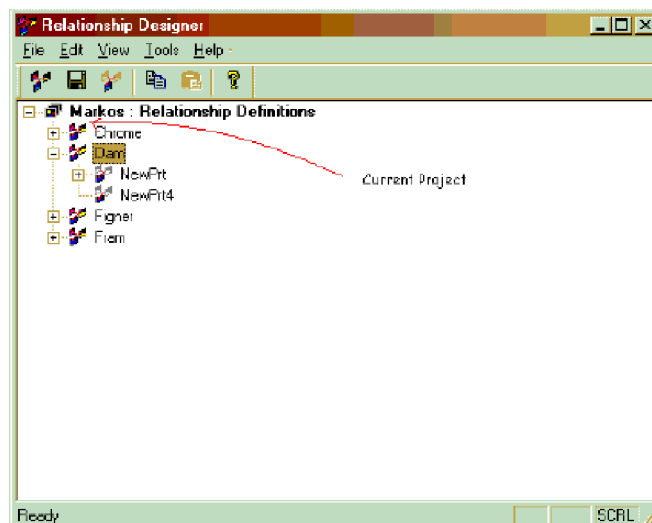


Figure 113. Relationship definition list

The Participant Types window shows a list of available data types in the current project that you can associate with a participant.

Figure 114 shows the main window of Relationship Designer, with both the Relationship Definition list and the Participant Types window.

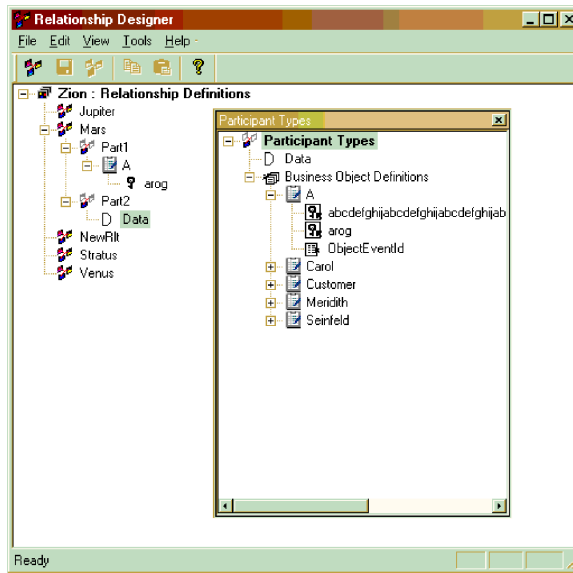


Figure 114. Relationship Designer main window

## Customizing the main window

Relationship Designer allows you to customize its main window by:

- “Selecting windows to display” on page 240
- “Floating a dockable window” on page 241

### Selecting windows to display

When you first open Relationship Designer, only the relationship definition list displays in the main window. The Participant Types window does not display. You can customize the appearance of the main window with options from the View pull-down menu. Table 82 describes the options of the View menu and how they affect the appearance of the Relationship Designer main window.

Table 82. View menu options for main window customization

View menu option	Element displayed
Participant Types	The Participant Types window displays
Toolbar	The Standard toolbar, which provides the main functionality for Relationship Designer
Status Bar	A single-line pane in which Relationship Designer displays status information

**Tip:** When a menu option appears with a check mark to the left, the associated element displays. To turn off the display of the element, select the associated menu option. The check mark disappears to indicate that the element does not currently

display. Conversely, you can turn on the display of an undisplayed element by selecting the associated menu option. In this case, the check mark appears beside the displaying element.

### Floating a dockable window

Relationship Designer supports the following features of the main window as dockable windows:

- Standard toolbar
- Participant Types window

**Tip:** 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.

## Using Relationship Designer functionality

You can access Relationship Designer’s functionality using any of the following:

- Pull-down menus
- Context menu
- Toolbar buttons
- Keyboard shortcuts

### Relationship Designer pull-down menus

Relationship Designer provides the following pull-down menus:

- File menu
- Edit menu
- View menu
- Tools menu
- Help menu

The following sections describe the options of each of these menus. Keyboard shortcuts are available for some of these options, as indicated.

**Functions of the File menu:** The File pull-down menu of Relationship Designer provides the options shown in Table 83. Except for the Switch to Project option, all File menu options affect objects in the current project.

*Table 83. File menu options in Relationship Designer*

File menu option	Description	For more information
New (Ctrl+N)	Creates a new relationship definition	“Creating a relationship definition” on page 243
Switch to Project (Ctrl+S)	Lists other projects	“Working with projects” on page 238
Save	Saves the current relationship definition to a file	“Creating a relationship definition” on page 243
Save All	Saves all open relationship definitions	N/A
Add Participant Definition	Adds a new participant definition to the current relationship definition	“Creating a relationship definition” on page 243

**Functions of the Edit menu:** The Edit pull-down menu of Relationship Designer provides the following options:

- Rename—renames the relationship definition
- Copy (Ctrl+C)—Copies the current relationship definition.
- Paste (Ctrl+V)—Pastes the copied relationship definition.
- Cut (Ctrl+X)—Deletes the current relationship definition.
- Advanced Settings—Displays the Advanced Settings window.

**Functions of the View menu:** The View pull-down menu of Relationship Designer displays the following options:

- Participant Types—Displays the Participant Types window.
- Expand Tree—Displays the members of the current level of the relationship definition list (same as clicking on the plus symbol beside the name of the level).
- Collapse Tree—Condenses the current level of the relationship definition list so that its members do not display (same as clicking on the minus symbol beside the name of the level).
- Toolbar—When enabled, displays the Standard toolbar.
- Status Bar—When enabled, displays a single-line status message at the bottom of the main window.

For information on the View menu options that control display, see “Selecting windows to display” on page 240.

**Tools menu functions:** The Tools pull-down menu of Relationship Designer provides options to start each of the WebSphere InterChange Server tools:

- Relationship Manager
- Process Designer
- Map Designer
- Business Object Designer

**Help Menu functions:** Relationship Designer provides a standard Help menu with the following options:

- Help Topics (F1)
- Documentation
- About Relationship Designer

**Context menu:** The Context menu is a shortcut menu that is available, by right-clicking from numerous places. A menu opens that contains useful commands, which change depending on the location you click.

### Relationship Designer Standard toolbar

Relationship Designer provides a Standard toolbar for common tasks you need to perform. This toolbar is dockable; that is, you can detach it from the palette of the main window and float it over the main window or the desktop.

**Tip:** To identify the purpose of each toolbar button, roll over each button with your mouse cursor.

Figure 115 shows the Relationship Designer Standard toolbar.



Figure 115. Relationship Designer Standard toolbar

The following list provides the function of each Standard toolbar button, left to right:

- New Relation
- Save Relation
- New Participant
- Copy
- Paste
- Help

---

## Creating a relationship definition

Perform the following steps to create a relationship definition:

1. Create a relationship name by doing one of the following:
  - From the File menu, select New Relationship Definition
  - Use the keyboard shortcut of Ctrl+N.
  - In the Standard toolbar, click the New Relation button.
2. Name the icon for the relationship definition.

**Rule:** Relationship definition names can be up to 8 characters long, can contain only letters and numbers, and must begin with a letter.
3. Create a participant definition for each business object to be related.

To do this, select the relationship definition name and perform one of the following actions:

  - From the File menu, select Add Participant Definition.
  - In the Standard toolbar, click the New Participant button.
4. For each participant definition, name the icon for the participant definition.

**Rule:** Participant definition names can be up to 8 characters long, can contain only letters and numbers, and must begin with a letter.
5. Associate a data type with each participant by dragging the type from the Participant Types window onto the participant definition.

**Tip:** To display the Participant Types window, select Participant Types from the View menu.

  - To associate a business object data type, drag the business object definition from the Participant Types window.

The participants in an identity relationship use business object definitions as their participant type. For more information, see “Defining identity relationships” on page 244.
  - To associate a Java data type, drag the Data participant type from the Participant Types window.

In the relationship definition, the Data participant type represents *all* data types other than business object types. The participants in a lookup relationship use Data as their participant type. For more information, see “Defining lookup relationships” on page 246.
6. For participant types that are business object definitions, add or change the attributes to associate with the participant.

The attributes you select become the basis on which the business objects are related.
7. Save the relationship definition by doing one of the following:
  - From the File menu, select Save Relationship Definition.
  - Use the keyboard shortcut of Ctrl+S.

- In the Standard toolbar, click the Save Relation button.
8. Before executing a map that uses the relationship definition, perform the following steps:
    - a. Activate the relationship. After the relationship is deployed to InterChange Server, this new relationship is *not* activated. However, for the Mapping API methods to be able to access the relationship tables, a relationship table must be active. To activate the relationship, click the relationship name in System Manager and select the Start option from the Component menu.
    - b. Compile and deploy the map that uses the relationship.
 

**Result:** If the map is deployed and compiled successfully in InterChange Server, InterChange Server creates the executable map code and activates the map. For more information, see “Compiling a map” on page 84.

**Restrictions:**

1. IBM supports creation of relationship tables only in those databases and platforms supported for the InterChange Server repository.
2. If you create or make a change to a relationship definition, you must first stop the relationship through the System Manager Relationship menu, make the change to the relationship, and then restart the relationship.

---

## Defining identity relationships

An *identity relationship* establishes an association between two or more business objects on a one-to-one basis. That is, for a given relationship instance, there can be only one instance of each participant. You typically create an identity relationship to transform the key attributes in a business object, such as customer or product ID. For more background information, see “Identity relationships” on page 225.

InterChange Server supports the kinds of identity relationships shown in Table 84..

Table 84. Kinds of Identity relationships

Identity relationship type	Description	For more information
Simple identity relationship	Relates two business objects through a single key attribute	“Using simple identity relationships” on page 263
Composite identity relationship	Relates two business objects through a composite key (made up of more than one attribute)	“Using composite identity relationships” on page 274

## Steps for defining identity relationships

To define an identity relationship using Relationship Designer, perform the following steps:

1. Create a relationship definition and the participant definitions by following steps 1-4 in “Creating a relationship definition” on page 243.
 

**Guideline:** Create a participant definition for each business object to be related. Identity relationships require participants for the generic business object as well as the application-specific business objects.
2. Associate a business object with each participant definition by dragging the business object definition from the Participant Types window onto the participant definition. You can release the drag button when the plus symbol (+) appears in the Relationship Designer main window. For information on how to open the Participant Types window, see step 5 in “Creating a relationship definition” on page 243.

For identity relationships, the participant type is a business object. Every identity relationship has a participant with a participant type of the generic business object plus one participant for each application-specific business object.

3. For each business object that you associate with a participant definition, add the attributes that relate the business object with the other participants.

To do so, expand the associated business object in the Participant Types window, select an attribute, and drag it onto the business object in the main Relationship Designer window. The attributes you select become the basis of the relationship between the business objects.

For identity relationships, the attributes are usually the key attributes of each business object definition. The type of the key determines the kind of identity relationship:

- For a single key, use a simple identity relationship. Each participant can consist of only one attribute: the unique key of the business object. For more information, see “Steps for creating the child relationship definition” on page 273.
  - For a composite key, use a composite identity relationship. Specify a composite key by adding each key attribute in the order in which it appears in the composite key. Each participant can contain several attributes: usually, the unique key from the parent business object and at least one attribute from the child business object (within the parent business object). When deployed to the server, the relationship is saved in a table, the name of which is the concatenation of the attributes in the order in which they appear in the participant definition. For more information, including the index size limitations of some databases, see “Creating composite identity relationship definitions” on page 274.
4. Highlight the relationship definition name and select Advanced Settings from the Edit menu.

Initially, the Advanced Settings window displays the relationship definition settings, as Figure 117 on page 249 shows.

- a. Modify the relationship definition settings as follows:

- Under Relationship type, select the Identity box.

**Result:** This setting tells InterChange Server to process the relationship as an identity relationship by setting a uniqueness constraint on the relationship instance ID and the key attributes for each participant. This action guarantees a one-to-one correspondence between all participants in each relationship instance.

- If you want the relationship tables to reside in a database other than the default database (the WebSphere business integration system repository, by default), enter the appropriate database information in the DBMS Settings area of the window. For more information, see “Advanced settings for relationship definitions” on page 249.

- b. Modify the advanced settings for the participant definition.

- In the object browser of the Advanced Settings window, expand the relationship definition and highlight the participant definition that represents the generic business object to display the participant definition settings (see Figure 118 on page 250). Select the box labeled IBM WBI-managed.

**Result:** This action tells Relationship Designer *not* to create relationship tables for the generic business object. When you maintain the relationship with the `maintainSimpleIdentityRelationship()` method, the WebSphere



business integration system uses the relationship instance IDs stored in the application-specific relationship tables to transform the relationship attributes.

- If you want to customize the name for this participant’s relationship table or stored procedure, enter the name in the appropriate field in the window. For more information, see “Advanced settings for participant definitions” on page 250.
- c. Click OK to close the Advanced Settings window.
5. Save the relationship definition as described in steps 7-8 in “Creating a relationship definition” on page 243.

## Relating child business objects

When you create identity relationships, the business objects you are relating often have child business objects. For instance, some customer business objects have child business objects for storing address information. A child business object can participate in the kinds of relationships that Table 85 shows.

Table 85. Relationships for child business objects

Condition of child business object	Kind of relationship	For more information
The key for the child business object <i>uniquely</i> identifies the child beyond the context of its parent	Simple identity relationship	“Coding a child-level simple identity relationship” on page 273
The key for the child business object does <i>not</i> uniquely identify it beyond the context of its parent	Composite identity relationship	
To maintain the child business objects during an Update operation as part of the identity relationship	Parent/child relationship	“Managing child instances” on page 282

If the child is a multiple-cardinality child business object, you can change the index to make the participant reference a specific child. To do so, select the child’s key attribute, right-click, and select Change Index from the Context menu. If the source and destination children in a map correspond one to one, the index is not significant and you do not need to change it. However, if the map transforms the children in any other way, you can enter a specific index number. For example, if the child business objects represent addresses and the third source address corresponds to the first destination address, you can change the indexes to 2 and 0, respectively.

## Defining lookup relationships

A *lookup relationship* associates data that is equivalent across business objects but may be represented in different ways. In this case, given a value in one business object, the relationship can look up its equivalent in the relationship tables for another business object. The most common example of attributes that might require lookups are codes (EmployeeType, PayLevel, OrderStatus) and abbreviations (State, Country, Currency). For more background information, see “Lookup relationships” on page 224.

When you create a relationship definition for a lookup, you add a participant definition for each business object that contains the attributes you want to relate. However, you do not associate the actual business object definitions or attribute names with the participant definitions. Instead, you specify Data as the participant type for each participant definition.



## Steps for defining lookup relationships

To define a lookup relationship using Relationship Designer, perform the following steps:

1. Create a relationship definition and the participant definitions by following Steps 1-4 in “Creating a relationship definition” on page 243.

**Tip:** Create a participant definition for each business object to be related.

2. For each participant definition, specify Data as the participant type by dragging the Data participant type from the Participant Types window onto the participant definition.

In the relationship definition, the Data participant type represents *all* data types other than business object types. When you create the map and work with instances of the relationship using methods in the Relationship, IdentityRelationship, and Participant classes, you can use data of any of the supported Java data types, such as String, int, long, float, double, or boolean.

3. Make a note of the table name for storing the lookup values for each participant definition. You need to know the table name so you can populate the tables with the lookup values for each participant definition. Or, if you already have tables containing the lookup values, you can replace the generated table name with your own table name.

To retrieve the table names for each participant definition in the relationship definition, or to specify your own table names:

- a. Select the participant definition and select Advanced Settings from the Edit menu.

**Result:** The Advanced Setting dialog box appears showing the storage settings for that participant. See “Specifying advanced relationship settings” on page 249 for more information on these settings.

- b. Write down the storage settings for the participant, or overwrite the settings with your own table information.

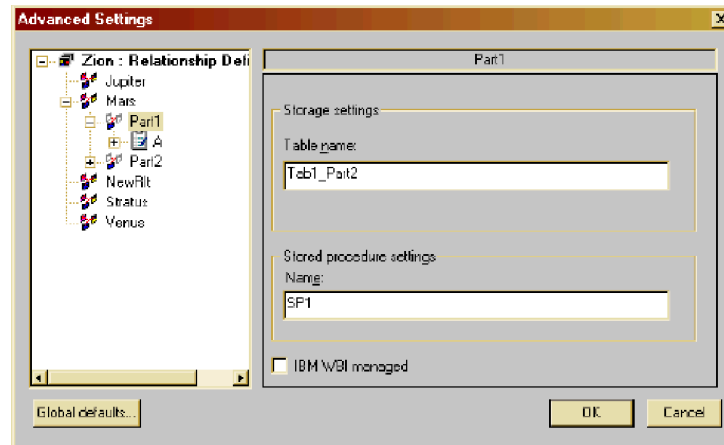


Figure 116. Advanced Settings dialog

- c. Repeat step 3a and step 3b for each participant definition.
  - d. Click OK to close the Advanced Settings dialog box.
4. Save the relationship definition as described in steps 7-8 in “Creating a relationship definition” on page 243.

**Tip:** To create the relationship tables, select the Create Schema box in the Deploy Project dialog in System Manager. For more information about when to create the run-time schema, see “Creating the relationship table schema” on page 248.

5. Using the information you gathered in step 3, populate the relationship tables with the lookup values for each participant, or add your own tables of lookup values to the database. For more information, see “Populating lookup tables with data” on page 259.

---

## Creating the relationship table schema

For each relationship definition you create, InterChange Server uses the following database objects to maintain the run-time data for instances of the relationship:

- Tables in the relationship database hold the data of the relationship instances.
- Stored procedures in the relationship database maintain the relationship tables.

---

## Copying relationship and participant definitions

To create a new relationship definition that is similar to an existing one, you can copy the existing definition and modify it to suit your needs. You can also copy a participant definition from a relationship definition and paste it into the same relationship definition or into another one.

### Steps for copying relationship definitions in the current project

To copy a relationship definition, perform the following steps:

1. Select the relationship definition you want to copy (for example, CustToClient) and select Save Relationship Definition from the File menu.
2. Select the relationship definition you want to copy and select Copy from the Edit menu.
3. Select the Project name (root tree node) and select Paste from the Edit menu.  
**Result:** Relationship Designer creates a new relationship definition with a name of Copy of CustToClient. The definition name appears in edit mode.
4. Enter a new name for the relationship definition, and then press Enter.
5. To save the new definition to the repository, select Save Relationship Definition from the File menu (or use the keyboard shortcut of Ctrl+S).

**Tip:** To copy a relationship definition from one InterChange Server to another, use the repos\_copy command. The repos\_copy command copies objects into and out of the InterChange Server repository.

### Steps for copying participant definitions in the current project

To copy a participant definition, perform the following steps:

1. Select the relationship definition to which the participant definition you want to copy belongs and select Save Relationship Definition from the File menu.
2. Select the participant definition you wish to copy and select Copy from the Edit menu.
3. Select the relationship definition to which you want to copy the participant definition and select Paste from the Edit menu.

**Result:** Relationship Designer creates a new participant definition with a name of Copy. The definition name appears in edit mode.

4. Enter a new name for the participant definition, and then press Enter.

---

## Renaming relationship or participant definitions

You can rename a relationship or participant definition before you save it to the repository. To change a definition's name after you have saved it, you must copy the definition to a new name and delete the old name. For help copying definitions, see "Copying relationship and participant definitions" on page 248.

---

## Specifying advanced relationship settings

For each relationship definition you create, Relationship Designer maintains advanced settings that affect the storage and processing of the relationship instance data.

**Note:** If you change any database-related setting, such as a login account name, password, or a table name after creating the relationship table schemas, you must re-create the relationship table schemas using System Manager for your changes to take effect.

To view or change the settings, select Advanced Settings from the Edit menu. In the Advanced Settings dialog, the settings that appear on the right side differ depending on which of the following items you have selected on the left:

- Relationship definition
- Participant definition
- Attribute

## Advanced settings for relationship definitions

To view or change the settings for a relationship definition, select the relationship name. The following illustration shows an example of the advanced settings at this level:

Select the relationship definition name to view or change its settings.

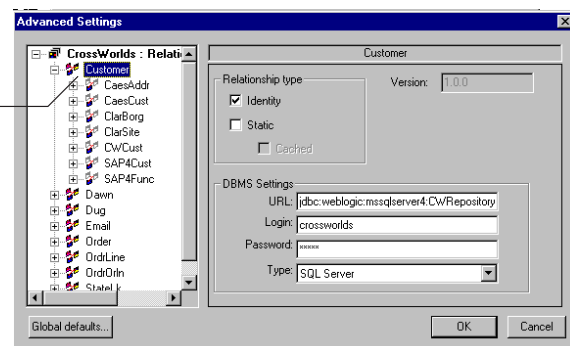


Figure 117. Advanced settings for a relationship definition

Table 86 summarizes the settings available for relationship definitions. Default values for the DBMS settings come from the Global Default Settings dialog box described in "Global default settings" on page 251.

Table 86. Summary of advanced settings for relationship definitions

Setting	Description
Relationship type	

Table 86. Summary of advanced settings for relationship definitions (continued)

Setting	Description
Dynamic (identity)	When this option is enabled, the relationship is a dynamic (identity) relationship. For more information, see “Defining identity relationships” on page 244.
Static (lookup)	When this option is enabled, the relationship is a static (lookup) relationship. For more information, see “Defining lookup relationships” on page 246.
Cached	When the Static field is enabled, this field is enabled. Select this field to have the relationship tables cached in memory. For more information, see “Optimizing a relationship” on page 253.
Version	This field is read-only. Versions for relationship definitions are not supported in this release.
DBMS Settings	
URL	The JDBC path where the relationship tables for this relationship definition are located. The default location for all relationship tables is specified in Global Default Settings (see 251).
Login	The user name for logging in to the relationship database.
Password	The password for logging in to the relationship database.
Type	The relationship database type, such as SQL Server or DB2.

**Note:** If you specify a database for the relationship tables that is different from the InterChange Server’s repository database, you might need to increase the setting for the maximum number of connection pools that the server can create. The server configuration parameter that specifies the number of connection pools is MAX\_CONNECTION\_POOLS. The default value is 10.

## Advanced settings for participant definitions

To view or change the settings for participant definitions, select the participant definition name. The following illustration shows an example of the advanced settings at this level:

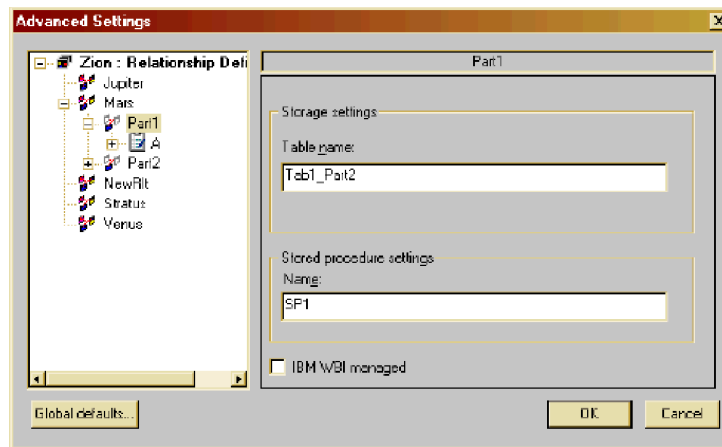


Figure 118. Advanced settings for a participant definition

Table 87 summarizes the settings available for participant definitions.

Table 87. Summary of advanced settings for participant definitions

Setting	Description
Table name	Name of the relationship table in the relationship database containing the relationship data for this participant instance.  <b>Rule:</b> DB2 relationship databases can only use a maximum of 17 characters in the relationship table names. Although table names do <i>not</i> have a limit in DB2, index names do. Because Relationship Designer generates index names for the relationship tables based on their table names, relationship table names for a DB2 database must be 17 characters or less.
Stored procedure name	Name of the stored procedure that maintains the relationship table.
IBM ICS managed	If selected, prevents relationship tables from being created for this participant. Select this setting only when: <ul style="list-style-type: none"><li>• The business object associated with this participant definition is a generic business object.</li><li>• There is only one attribute associated with the participant and it is a key attribute.</li></ul>

## Advanced settings for attributes

To view or change the advanced settings for an attribute, select the attribute. The following illustration shows an example of the advanced settings:

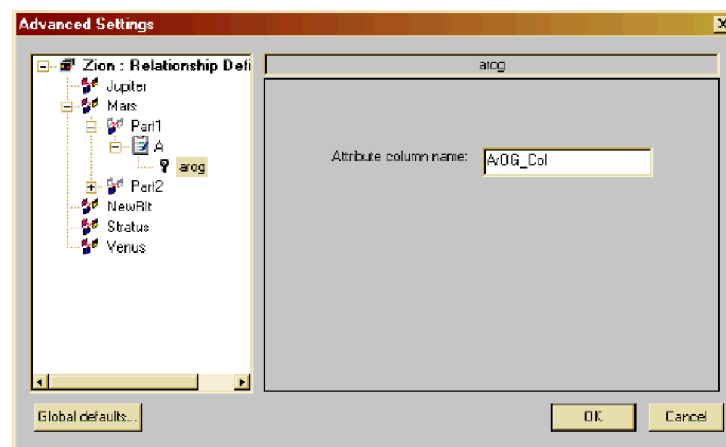


Figure 119. Advanced settings for attributes

For attributes, the only setting available is the attribute column name. The column name is the name of the column in the relationship table that contains the values for the selected attribute. It is typically the same as the attribute name. You might want to change the column name if you are using tables you created instead of the default tables that the Relationship Designer creates.

## Global default settings

When you save a new relationship definition and create the relationship table schemas, Relationship Designer must know the location of the database for the relationship tables, the type of database, and how to access the database with a valid user name and password. Relationship Designer maintains default values for

these settings, which it uses for all new relationship definitions you create. Once a relationship definition is created, these settings are stored with the relationship definition, and you can change the settings for each relationship definition individually.

By default, the database name and access information is the same one used by the InterChange Server repository. If you want to store your relationship tables in another location, you can modify the global settings.

### Steps for viewing or changing the global default settings

To view or change the global default settings, perform the following steps:

1. In Relationship Designer, select Advanced Settings from the Edit menu.

**Result:** The Advanced Settings dialog box appears.

2. Click the Global defaults button.

**Result:** The Global Default Settings dialog box appears.

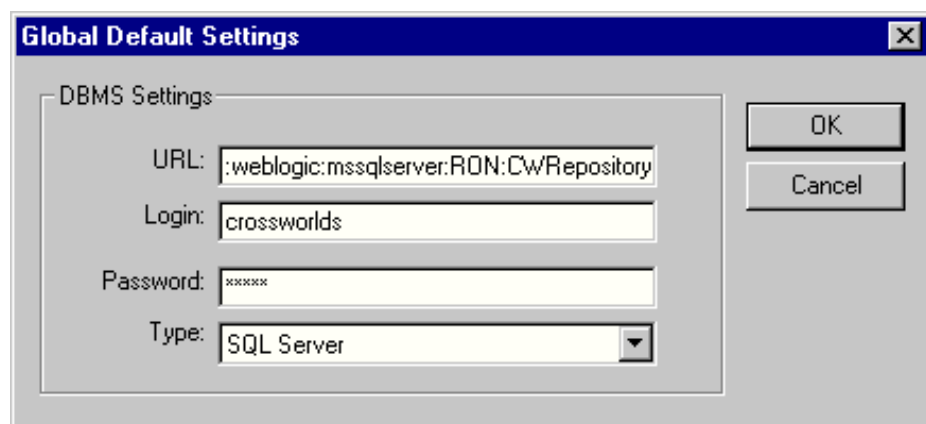


Figure 120. Global Default Settings dialog

Table 88 describes the global default settings for relationships.

Table 88. Relationship global default settings

Setting	Description
URL	The JDBC path where the relationship database is located. The default is the InterChange Server's repository database.
Login	The user name for logging in to the relationship database.
Password	The password for logging in to the relationship database.
Type	The relationship database type, such as SQL Server or DB2.

**Note:** If you specify a database for the relationship tables that is different from the InterChange Server's repository database, you might need to increase the setting for the maximum number of connection pools that the server can create. The server configuration parameter that specifies the number of connection pools is MAX\_CONNECTION\_POOLS. The default value is 10.

3. When you are finished viewing or making changes, click OK to save or Cancel to exit without saving.

**Note:** Changes that you make to the global default settings apply only to new relationship definitions. They do not affect existing relationships. If you

want to change the settings for an existing relationship, see “Specifying advanced relationship settings” on page 249.

---

## Deleting a relationship definition

You can delete a relationship definition listed in the main window of Relationship Designer in either of the following ways:

- Highlight the definition and select Delete from the Edit menu.
- Right-click on the definition and select Delete.

---

## Optimizing a relationship

By default, each relationship’s relationship tables are stored in the relationship database. Each time a relationship retrieves or modifies run-time data, it uses SQL statements to access this database. If the relationship tables are accessed frequently, these accesses can have a significant impact on performance in terms of CPU usage and InterChange Server resources. As part of the design of a relationship, you can determine whether to cache these relationship tables into memory.

To make this decision, you need to determine how frequently the relationship’s run-time data changes. The WebSphere business integration system allows you to categorize your relationship in one of two ways:

- Dynamic relationship—a relationship whose run-time data changes frequently; that is, its relationship tables have frequent Insert, Update, or Delete operations. All relationships are dynamic by default.
- Static relationship—a relationship whose run-time data undergoes very minimal change; that is, its relationship tables have very few Insert, Update, or Delete operations. For example, because lookup tables store information such as codes and status values, their data very often is static. Such tables make good candidates for being cached in memory.

**Note:** The WebSphere InterChange Server System Manager categorizes relationships into these same two categories. When you expand the Relationships folder, System Manager displays two subfolders: Dynamic and Static.

You define whether a relationship is dynamic or static from the Advanced Setting dialog for the relationship definition. The following sections summarize how to define a dynamic and static relationship from this dialog. For information on how to display the Advanced Setting dialog, see “Specifying advanced relationship settings” on page 249.

### Defining a dynamic relationship

For a dynamic relationship, InterChange Server accesses the run-time data from its relationship tables in the relationship database. By default, InterChange Server assumes a relationship is dynamic. Therefore, you do not have to perform any special steps to define a dynamic relationship:

- For an identity relationship, click the Dynamic (identity) field on the Advanced Settings dialog, as described in “Defining identity relationships” on page 244.
- For a lookup relationship, make sure the Dynamic (identity) field is *not* selected, as described in “Defining lookup relationships” on page 246.

**Note:** For a dynamic relationship, do *not* click the Static (lookup) or Cached field on the Advanced Settings dialog.

System Manager lists all dynamic relationships in the folder labeled Dynamic under the Relationships folder.

## Defining a static relationship

For a static relationship, InterChange Server can access the run-time data from cached relationship tables. With caching enabled for the static relationship, InterChange Server stores a copy of the relationship tables in memory. When making the decision to cache relationship tables, try to balance the following conditions:

- Performance usually improves if you let InterChange Server cache the relationship tables in memory.

In this case, the server does not need to use SQL statements to access the relationship database for the run-time data. Instead, it can access memory for this data, which is much faster. If the run-time data for a static relationship is not currently in memory, InterChange Server reads the appropriate relationship tables from the database into memory when the data is first accessed. For future accesses, InterChange Server uses the cached version of the tables.

However, once the table is read into memory, InterChange Server must maintain consistency between the relationship tables in the database and the cached tables. For Update, Insert, and Delete operations, InterChange Server must modify *both* the database tables *and* the cached tables. This double update can be very performance intensive. When you determine whether to cache a relationship's tables, consider the expected lifetime and refresh rate of the data.

- Memory usage increases when relationship tables are cached in memory. The amount of memory used is roughly equivalent to the size of all in-memory tables.

**Recommendation:** You should not cache a relationship table that contains more than 1000 rows.

**Important:** InterChange Server does *not* check for excessive memory usage. You must ensure that memory usage remains within the limits that your system imposes.

To define a static relationship, display the Advanced Settings dialog (see Figure 117) for the relationship definition and set the Static field from this dialog as follows:

- For an identity relationship, enable both the Dynamic (identity) and Static (lookup) fields. For more information on the use of the Dynamic (Identity) field, see "Defining identity relationships" on page 244.
- For a lookup relationship, enable the Static (lookup) field, *not* the Dynamic (identity) field.

When the Static (lookup) field is enabled, the Advanced Settings dialog also enables the Cached field. The Cached field allows you to control when InterChange Server actually caches the relationship's table:

- When Cached is enabled, InterChange Server can cache the relationship tables for a static relationship. It caches *all* relationship tables involved in the relationship.
- When Cached is disabled, InterChange Server does not cache the relationship tables in memory. Instead, it uses the tables in the relationship database for future accesses.

You can only control caching for a relationship that is defined as static.



**Note:** After you change a relationship's static or cached state from the Advanced Settings dialog, make sure you save the relationship definition for the change to be stored in the project.

**Note:** You can modify the cached and reload relationship properties from the server component management view. To do this, right-click the static relationship and select the properties from the Context menu.

- **Cached**—controls caching of the relationship's tables.
- **Reload**—tells InterChange Server to reread the relationship's tables into memory.



---

## Chapter 8. Implementing relationships

Relationship attributes are those you transform using relationships. You do *not* transform relationship attributes by dragging from source attribute to destination attribute. Instead, you create a Custom transformation and customize the transformation rule for the destination relationship attribute using the function blocks in Activity Editor or write code for the destination relationship attribute using methods in the Relationship, IdentityRelationship, and Participant classes.

This chapter describes how to develop code within a map to implement the different kinds of relationships. It covers the following tasks.

**Note:** This chapter assumes that you have already created the relationship definitions for the relationships. For information, see Chapter 7, “Creating relationship definitions,” on page 237.

- “Implementing a relationship” on page 257
- “Using lookup relationships” on page 258
- “Using simple identity relationships” on page 263
- “Using composite identity relationships” on page 274
- “Managing child instances” on page 282
- “Setting the verb” on page 285
- “Performing foreign key lookups” on page 290
- “Maintaining custom relationships” on page 295
- “Writing safe relationship code” on page 297
- “Executing queries in the relationship database” on page 299
- “Loading and unloading relationships” on page 309

---

### Implementing a relationship

Once you have created a relationship definition within Relationship Designer you are ready to implement the relationship within the map. For instructions on creating relationship definitions, see Chapter 7, “Creating relationship definitions,” on page 237.

To implement a relationship, you can use the relationship function blocks in the map’s destination object or add Mapping API methods to the code of attributes in the map’s destination object.

Table 89 shows the function blocks to use.

*Table 89. Relationship function blocks*

---

Kind of relationship	Function block	For more information
Lookup	General/APIs/Relationship/Retrieve Instances General/APIs/Relationship/Retrieve Participants	“Using lookup relationships” on page 258
Simple Identity	General/APIs/Identity Relationship/Maintain Simple Identity Relationship General/APIs/Identity Relationship/Maintain Child Verb	“Using simple identity relationships” on page 263

---

Table 89. Relationship function blocks (continued)

Kind of relationship	Function block	For more information
Composite Identity	General/APIs/Identity Relationship/Maintain Composite Relationship General/APIs/Identity Relationship/Maintain Child Verb General/APIs/Identity Relationship/Update My Children (optional)	“Using composite identity relationships” on page 274
Custom	General/APIs/Relationship/Create Relationship General/APIs/Identity Relationship/Add My Children General/APIs/Relationship/Add Participant	

Table 89 shows the Mapping API methods that maintain the different kinds of relationships.

Table 90. Mapping API methods for relationships

Kind of relationship	Mapping API method	For more information
Lookup	retrieveInstances() retrieveParticipants()	“Using lookup relationships” on page 258
Simple Identity	maintainSimpleIdentityRelationship() maintainChildVerb()	“Using simple identity relationships” on page 263
Composite Identity	maintainCompositeRelationship() maintainChildVerb() updateMyChildren() (optional)	“Using composite identity relationships” on page 274
Custom	create() addMyChildren() addParticipant()	“Maintaining custom relationships” on page 295

When transforming relationship attributes, a map needs to know the calling context of the map. To determine the calling context, the map needs the following information from the map execution context:

- The map’s calling context, which is part of the map execution context  
For more information, see “Calling contexts” on page 190.
- The verb, which is part of the business object

These two factors tell the map what actions need to be taken on the relationship tables.

For the relationships in Table 89, the associated Mapping API methods perform the appropriate operations on the relationship tables. Therefore, these methods require that the calling context and business object verb be passed in as arguments.

## Using lookup relationships

A *lookup relationship* associates data that is equivalent across business objects but may be represented in different ways. The following sections describe the steps for using lookup relationships:

- “Creating lookup relationship definitions” on page 259
- “Populating lookup tables with data” on page 259
- “Customizing map transformations for a lookup relationship” on page 261

**Note:** For background information, see “Lookup relationships” on page 224..

## Creating lookup relationship definitions

Lookup relationship definitions differ from identity relationship definitions in that the participant types are *not* business objects but of the type Data (the first selection in the participant types list). For more information on how to create a relationship definition for a lookup relationship, see “Defining lookup relationships” on page 246.

**Example:** Suppose you create a lookup relationship called StatAdtp for the AddressType values. In Figure 121, each box represents a participant in the StatAdtp lookup relationship. Notice that each participant in this relationship is of type Data.

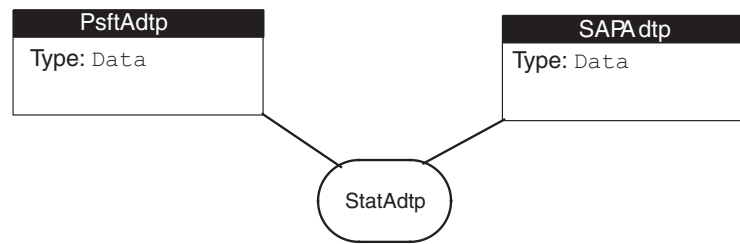


Figure 121. The StatAdtp lookup relationship definition

Because a lookup relationship does not indicate which attributes are being related, you can use one lookup relationship definition for transforming several attributes. In fact, you can use one lookup relationship definition for every attribute that requires a lookup, regardless of the business object being transformed. However, because only one set of tables is created for each relationship definition, using one relationship definition for all lookup relationships would make the tables large and hard to maintain.

A better strategy might be to create one lookup relationship definition per common unit of data, such as country code or status. This way, each set of relationship tables contains information related by meaning. Relationships defined this way are also more modular because you can add new participants, as you support new collaborations or applications, and reuse the same relationship definition. For instance, suppose you create a lookup relationship definition for country code to transform Clarify\_Site business objects to SAP\_Customer. Later on, if you add new collaborations or a new application, you can reuse the same relationship definition for every transformation involving a country code.

## Populating lookup tables with data

When you deploy the lookup relationship definition with the option Create Schema enabled, Interchange Server generates a relationship table (also called a *lookup table*) for each participant. Each lookup table has a name of the form: *RelationshipDefName\_ParticipantDefName*

When you deploy the StatAdtp relationship definition (see Figure 121) with the option Create Schema enabled, Interchange Server generates the following two lookup tables:

- StatAdtp\_PsftAdtp\_T
- StatAdtp\_SAPAdtp\_T

A lookup table contains a column for the relationship instance ID (INSTANCEID) and its associated participant instance data (data). Figure 122 shows the lookup tables

for the PsftAdtp and SAPAdtp participants in the StatAdtp lookup relationship. These two lookup tables use the relationship instance ID to correlate the participants. For example, the instance ID of 116 correlates the PsftAdtp value of Fired and the SAPAdtp value of 04.

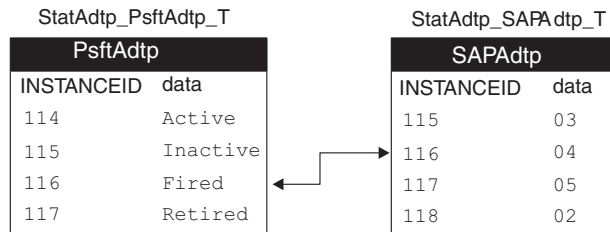


Figure 122. Relationship tables for the CustLkUp lookup relationship

Unlike relationship tables that hold data for identity relationships, lookup tables do *not* get populated automatically. You must populate these tables by inserting data into their columns. You can populate a lookup table in either of the following ways:

- Create a script that contains SQL INSERT statements to fill the lookup table with the desired data.
- Use Relationship Manager to add rows to the lookup table.

### Inserting participant instances with SQL

You can insert participant data into a lookup table with the SQL statement INSERT. This method is useful when you need to add many rows of data to the lookup table. You can create the syntax for one INSERT statement and then use the editor to copy and paste this line as many times as you have rows to insert. In each line, you only have to edit the data to be inserted (usually in a VALUES clause of the INSERT statement).

To use the INSERT statement, you must know the name of the lookup relationship table and its columns. Table 91 shows the column names in a lookup table.

Table 91. Columns of a lookup table

Column in lookup table	Description
INSTANCEID	The relationship instance ID.
data	The participant data
STATUS	Set to zero (0) when the participant is active
LOGICAL_STATE	Indicates whether the participant instance has been logically deleted (zero indicates “no”)
TSTAMP	Date of last modification for the participant instance.

**Attention:** When you use SQL statements to insert participant data into a lookup table, make sure you provide a value for the STATUS, LOGICAL\_STATE, and TSTAMP columns. All values are required for IBM WebSphere business integration tools to function correctly. In particular, omission of the TSTAMP value causes Relationship Manager to be unable to retrieve the participant data; if no timestamp value exists, Relationship Manager raises an exception.

**Example:** Suppose you want to add the participant data in to the relationship table that contains information for address type, shown in Table 92.

Table 92. Sample values for address type for PsftAdtp participant

INSTANCEID	STATUS	LOGICAL_STATE	TSTAMP	data
1	0	0	current date	Home
2	0	0	current date	Mailing

The following INSERT statements create the Table 92 participant data in the PstfAdtp lookup table:

```
INSERT INTO StatAdtp_PsftAdtp_T
    (INSTANCEID, STATUS, LOGICAL_STATE, TSTAMP, data)
VALUES (1, 0, 0, getDate(), 'Home')

INSERT INTO StatAdtp_PsftAdtp_T
    (INSTANCEID, STATUS, LOGICAL_STATE, TSTAMP, data)
VALUES (2, 0, 0, getDate(), 'Mailing')
```

**Note:** The preceding INSERT syntax is compatible with the MicroSoft SQL Server 7.0. If you are using another database server for your relationship table, make sure you use INSERT syntax compatible with that server.

### Inserting participant instances with Relationship Manager

Relationship Manager is an IBM WebSphere business integration tool that graphically displays run-time data in a relationship table. Relationship Manager is useful when you only need to add a few rows to the lookup table.

## Customizing map transformations for a lookup relationship

Once you have created the relationship definition and participant definitions for the lookup relationship, you can customize the map transformation rule for performing the lookups. For information on customizing lookup relationships in Activity Editor, see “Example 3: Using Static Lookup for conversion” on page 157.

Table 93 shows the Mapping API methods needed to implement a lookup relationship. This table also lists in which map the API call is needed.

Table 93. Mapping API methods for lookup relationships

Step in lookup relationship	Map	Mapping API method
Obtain relationship instance ID for the participant data from the source business object. The instance ID is saved in the generic business object.	Inbound map	retrieveInstances()
Obtain participant instances for the relationship instance ID from the generic business object. The participant data is saved in the application-specific business object.	Outbound map	retrieveParticipants()

**Tip:** The retrieveInstances() and retrieveParticipants() methods do *not* populate the lookup tables. They assume that the participant data already exists in the tables. Make sure you populate lookup tables before you run a map that contains a lookup relationship. For more information, see “Populating lookup tables with data” on page 259.

## Coding the inbound map

You call the `retrieveInstances()` method in the inbound map to retrieve relationship instance IDs for the participant data in the source business object. The following piece of code performs a lookup in the inbound map:

```
String addrtype = ObjSrcObj.getString("SrcAttr");
int[] generic_ids = Relationship.retrieveInstances(
    "RelationshipDefName", "ParticipantDefName", dataFromSrcObj);
if (generic_ids != null && generic_ids.length > 0)
{
    ObjDestObj.setWithCreate("DestAttr", generic_ids[0]);
}

else
{
    throw new MapFailureException(
        logError("No generic instance ID for lookup found");
        "No generic instance ID for lookup found");
}
```

**Tips:** Keep the following tips in mind when coding the `retrieveInstances()` method:

- The `retrieveInstances()` method does *not* raise an exception if it does not find matching instance IDs for a particular participant data. To raise an exception if no matching instances IDs are found, the preceding code fragment checks the returned instance IDs (`generic_ids`) for a null value *before* it sets the destination business object with `setWithCreate()`.
- The `retrieveInstances()` method returns an *array* of relationship instance IDs. Usually, a lookup relationship is structured so that each given piece of participant data is associated with only one instance ID. However, `retrieveInstances()` does not assume such a one-to-one correspondence. It returns an array so it can return multiple relationship instance IDs.

## Coding the outbound map

You call the `retrieveParticipants()` method in the outbound map to retrieve relationship instance IDs for the participant data in the source business object. The following piece of code performs a lookup in the outbound map:

```
int addrtype = ObjSrcObj.getInt("SrcAttr");

if (addrtype != null)
{
    Participant[] psft_part = Relationship.retrieveParticipants(
        "RelationshipDefName", "ParticipantDefName", addrtype);
    if (psft_part != null && psft_part.length > 0)
        ObjDestObj.setWithCreate("DestAttr", psft_part[0].getString());
}
```

**Tips:** Keep the following tips in mind when coding the `retrieveParticipants()` method:

- If your outbound map cannot assume that the generic business object contains a relationship instance ID, you might want to check for a null-valued instance ID *before* calling `retrieveParticipants()`. The `retrieveParticipants()` method raises a `RelationshipRuntimeException` exception if it receives a null-valued instance ID.
- The `retrieveParticipants()` method returns an *array* of participant instances. Usually, a lookup relationship is structured so that each relationship instance ID is associated with only one piece of participant data. However, `retrieveParticipants()` does not assume such a one-to-one correspondence. It returns an array so it can return multiple participant instances.



---

## Using simple identity relationships

An identity relationship establishes an association between business objects or other data on a *one-to-one* basis. A simple identity relationship relates two business objects through a single key attribute. The following sections describe the steps for working with simple identity relationships:

- “Creating simple identity relationship definitions”
- “Accessing identity relationship tables”
- “Defining transformation rules for a simple identity relationship” on page 273

### Creating simple identity relationship definitions

Identity relationship definitions differ from lookup relationship definitions in that the participant types are business objects, *not* of the type Data (the first selection in the participant types list). For a simple identity relationship, the relationship consists of the generic business object and at least one application-specific business object. The participant type for a simple identity relationship is a business object for *all* participants. The participant attribute for every participant is a single key attribute of the business object. (For more information on how to create a relationship definition for a simple identity relationship, see “Defining identity relationships” on page 244.)

### Accessing identity relationship tables

To reference a simple identity relationship, define a Cross-Reference transformation rule between the application-specific business object and the generic business object. For more information, see “Cross-referencing identity relationships” on page 47.

**Example:** The `CustIden` relationship (see Figure 103) transforms a `SiteID` key attribute in a Clarify customer to a `RefID` key attribute in an SAP customer. It includes maps between the following objects:

- Inbound map: `Clarify_Site` to `Customer`  
Obtain from the `ClarCust` relationship table the relationship instance ID that is associated with the `SiteID` key value.
- Outbound map: `Customer` to `SAP_Customer`  
Obtain from the `SAPCust` relationship table the `RefID` key value that is associated with relationship instance ID.

Figure 123 shows how to use the `CustIden` relationship tables to transform a `SiteID` value of `A100` to a `RefID` value of `806`.

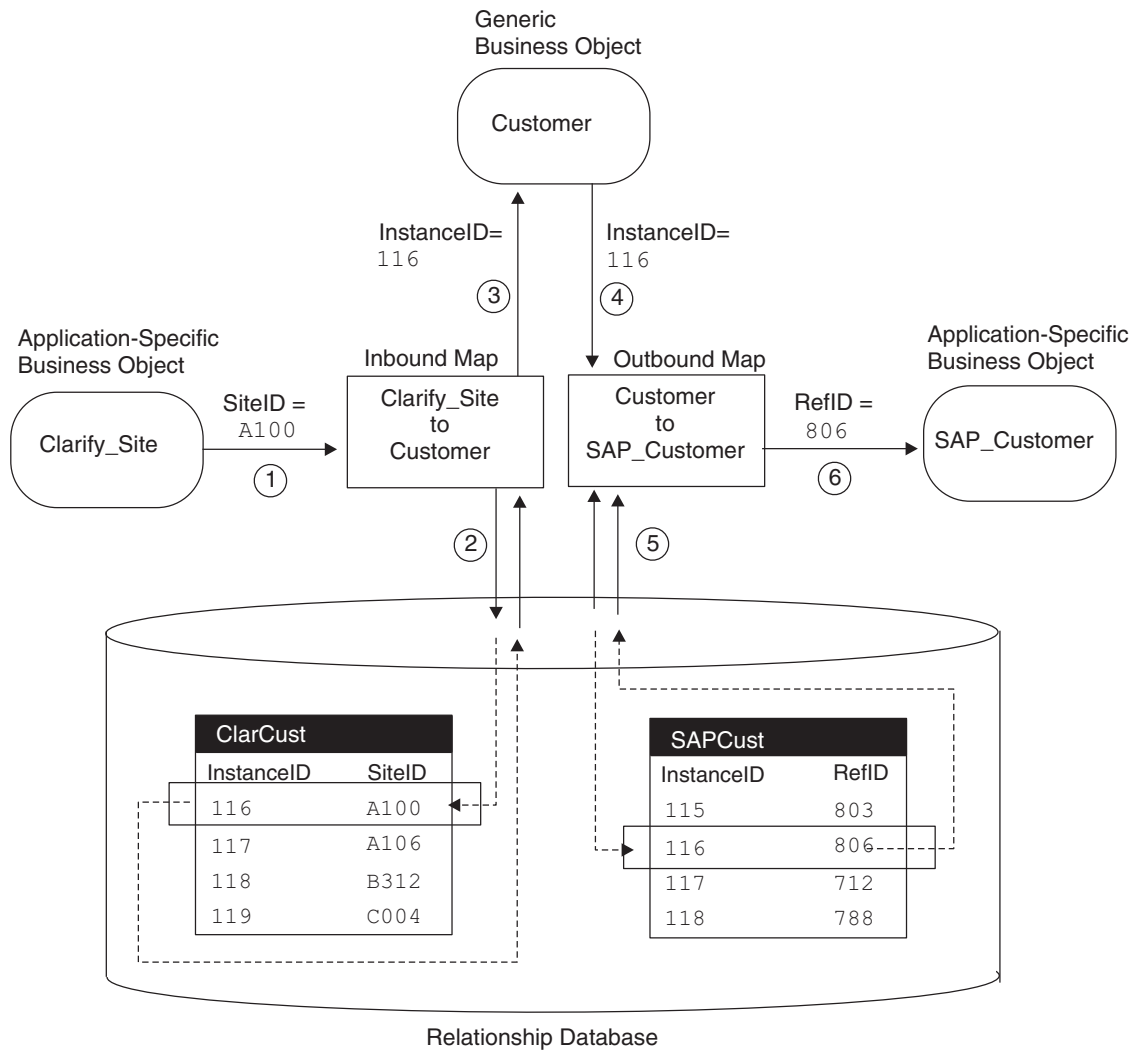


Figure 123. Using relationship tables to transform a SiteID to a RefID

The `maintainSimpleIdentityRelationship()` method must manage the relationship tables to ensure that related application-specific keys remain associated to a single relationship instance ID. At a high level, the Cross-Reference transformation rule generates code to do the following:

1. Perform validations on the arguments that are passed in. If an argument is invalid, the method throws the `RelationshipRuntimeException` exception. For a list of validations that the Java code generated by the Cross-Reference transformation performs, see the `maintainSimpleIdentityRelationship()` API in Chapter 21, "IdentityRelationship class," on page 437.
2. Take the appropriate actions to maintain the relationship tables based on the calling context, which includes the following factors:
  - The verb of the business object  
The Cross-Reference transformation obtains this verb from the source business object. For inbound maps, the source is the application-specific business object; for outbound maps, the source is the generic business object.
  - The value of the calling context  
The Cross-Reference transformation rule obtains the calling context from the map execution context automatically.

This transformation deals with the calling contexts shown in Table 94.

*Table 94. Calling contexts with maintainSimpleIdentityRelationship()*

Calling context	Description
EVENT_DELIVERY	A connector has sent an event from the application to InterChange Server (event-triggered flow).
ACCESS_REQUEST	An access client has sent an access request from an external application to InterChange Server
SERVICE_CALL_REQUEST	A collaboration is sending a business object down to the application through a service call request.
SERVICE_CALL_RESPONSE	A business object was received from the application as a result of a successful response to a collaboration service call request.
SERVICE_CALL_FAILURE	A collaboration's service call request has failed. As such, corrective action might need to be performed.
ACCESS_RESPONSE	The source business object is sent back to the source access client in response to a subscription delivery request.

The following sections discuss the behavior of the Cross-Reference transformation with each of the calling contexts in Table 94.

**Note:** For more information on calling maps within collaborations, see "Calling a native Map" section in the *Collaboration Development Guide*

### **EVENT\_DELIVERY and ACCESS\_REQUEST calling contexts**

When the calling context is EVENT\_DELIVERY or ACCESS\_REQUEST, the map that is being called is an inbound map; that is, it transforms an application-specific business object to a generic business object. A connector sends the EVENT\_DELIVERY calling context; an access client sends an ACCESS\_REQUEST calling context. In either case, the inbound map receives an application-specific business object as input and returns a generic business object as output. Therefore, the task for the Cross-Reference transformation is to obtain from the relationship table a relationship instance ID for a given application-specific key value.

For the EVENT\_DELIVERY and ACCESS\_REQUEST calling contexts, the Java code generated by the Cross-Reference transformation takes the following actions:

1. Locates the relationship instance in the relationship table that matches the given application-specific business object's key value. Table 95 shows the actions that the Java code generated by the Cross-Reference transformation takes on the relationship table based on the verb of the application-specific business object.
2. Obtains the instance ID from the retrieved relationship instance.
3. Copies the instance ID into the generic business object.

*Table 95. Actions for the EVENT\_DELIVERY and ACCESS\_REQUEST Calling Contexts*

Verb of application-specific business object	Action Performed by maintainSimpleIdentityRelationship()
Create	Insert a new entry into the relationship table for the application-specific business object's key value. If an entry for this key value already exists, retrieve the existing one; do <i>not</i> add another one to the table.

Table 95. Actions for the *EVENT\_DELIVERY* and *ACCESS\_REQUEST* Calling Contexts (continued)

Verb of application-specific business object	Action Performed by <code>maintainSimpleIdentityRelationship()</code>
Update	Retrieve the relationship entry from the relationship table for the given application-specific business object's key value. If an entry for this key value does <i>not</i> exist, add one to the table.
Delete	<ol style="list-style-type: none"> <li>Retrieve the relationship entry from the relationship table for the given application-specific business object's key value.</li> <li>Mark the relationship entry as "deactive".</li> </ol>
Retrieve	Retrieve the relationship entry from the relationship table for the given application-specific business object's key value. If an entry for this key value does <i>not</i> exist, throw a <code>RelationshipRuntimeException</code> exception.

For an identity relationship that supports the transformation of an AppA application-specific business object to AppB application-specific business object, Figure 124 shows how the Java code generated by the Cross-Reference transformation accesses a relationship table associated with the AppA participant when a calling context is *EVENT\_DELIVERY* (or *ACCESS\_REQUEST*) and the AppA application-specific business object's verb is either *Create* or *Update*.

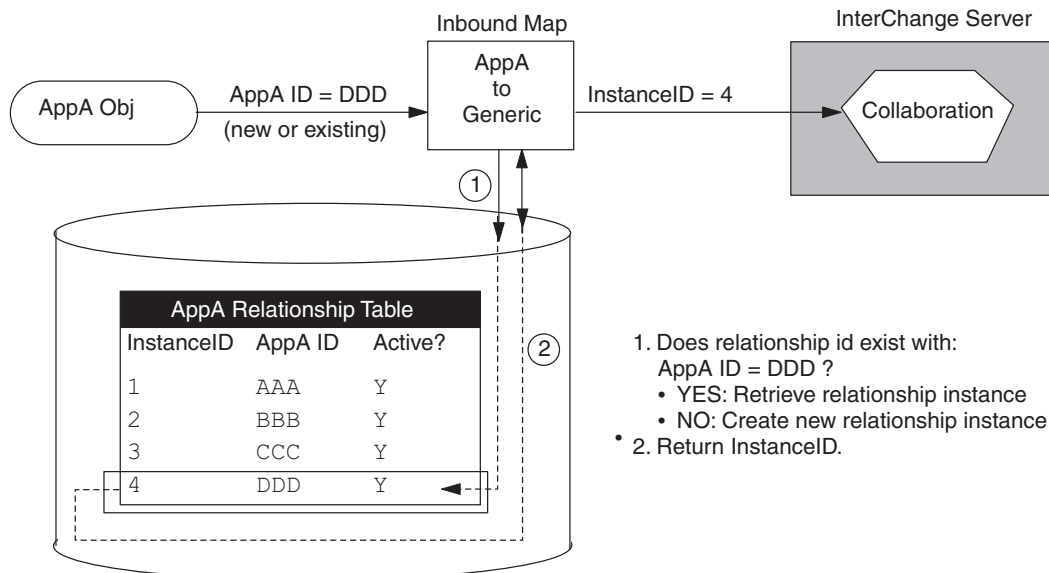


Figure 124. *EVENT\_DELIVERY* and *ACCESS\_REQUEST* with a create or update verb

For a calling context of *EVENT\_DELIVERY* (or *ACCESS\_REQUEST*) and an application-specific verb of either a *Create* or *Update*, Figure 125 shows the write that the Java code generated by the Cross-Reference transformation makes to the relationship table when no entry exists that matches the AppA application-specific key value.

Before Create			After Create		
AppA Relationship Table			AppA Relationship Table		
InstanceID	AppA ID	Active?	InstanceID	AppA ID	Active?
1	AAA	Y	1	AAA	Y
2	BBB	Y	2	BBB	Y
3	CCC	Y	3	CCC	Y
			4	DDD	Y

New Relationship Entry

Figure 125. The Write to the relationship table for a new relationship entry

For a calling context of `EVENT_DELIVERY` (or `ACCESS_REQUEST`) and an application-specific verb of Delete, Figure 126 shows the write that the Java code generated by the Cross-Reference transformation performs on the AppA relationship table.

Before Delete			After Delete		
AppA Relationship Table			AppA Relationship Table		
InstanceID	AppA ID	Active?	InstanceID	AppA ID	Active?
1	AAA	Y	1	AAA	Y
2	BBB	Y	2	BBB	N
3	CCDDDD	Y	3	CCC	Y
4		Y	4	DDD	Y

"Deleted" Row

Figure 126. The write to the relationship table for a delete verb

### SERVICE\_CALL\_REQUEST calling context

When the calling context is `SERVICE_CALL_REQUEST`, the map that is being called is an outbound map; that is, it transforms a generic business object to an application-specific business object. The outbound map receives a generic business object as input and returns an application-specific business object as output. Therefore, the task for the Cross-Reference transformation is to obtain from the relationship table an application-specific business object's key value for a given relationship instance ID *only* if the verb is Update, Delete, or Retrieve. The Cross-Reference transformation does *not* obtain the application-specific key value for a Create verb.

Table 96 shows the action that the Cross-Reference transformation takes on the relationship table based on the verb of the generic business object.

Table 96. Actions for the `SERVICE_CALL_REQUEST` calling context

Verb of generic business object	Action performed by the Cross-Reference transformation
Create	Take no action.
	When the calling context is <code>SERVICE_CALL_RESPONSE</code> , the method actually writes a new entry to the relationship table. For more information, see "SERVICE_CALL_RESPONSE calling context" on page 269.

Table 96. Actions for the SERVICE\_CALL\_REQUEST calling context (continued)

Verb of generic business object	Action performed by the Cross-Reference transformation
Update Delete Retrieve	<ol style="list-style-type: none"> <li>1. Obtain the generic business object's key value (the relationship instance ID) from the original-request business object in the map execution context.</li> <li>2. Retrieve the entry from the relationship table for the given generic business object's key value. If an entry for this key value does <i>not</i> exist, throw a RelationshipRuntimeException exception. If no participants are found when the verb is Retrieve, throw a CxMissingIDException exception.</li> <li>3. Obtain the application-specific key value from the retrieved relationship entry.</li> <li>4. Copy the application-specific key value into the application-specific business object.</li> </ol>

As Table 96 shows, when the verb is Create, the Java code generated by the Cross-Reference transformation does *not* write a new entry to the relationship table. It does not perform this write operation because it does not yet have the application-specific key value that corresponds to the relationship instance ID. When the connector processes the application-specific business object, it notifies the application of the need to insert a new row (or rows). If this insert is successful, the application notifies the connector, which creates the appropriate application-specific business object with a Create verb and the application's key value.

For the remaining verbs (Update, Delete, and Retrieve), the Java code generated by the Cross-Reference transformation performs a read operation on the relationship table. For an identity relationship that supports the transformation of an AppA application-specific business object to AppB application-specific business object, Figure 127 shows how the Cross-Reference transformation accesses a relationship table associated with the AppB participant when a calling context is SERVICE\_CALL\_REQUEST and the generic business object's verb is Update, Delete, or Retrieve.

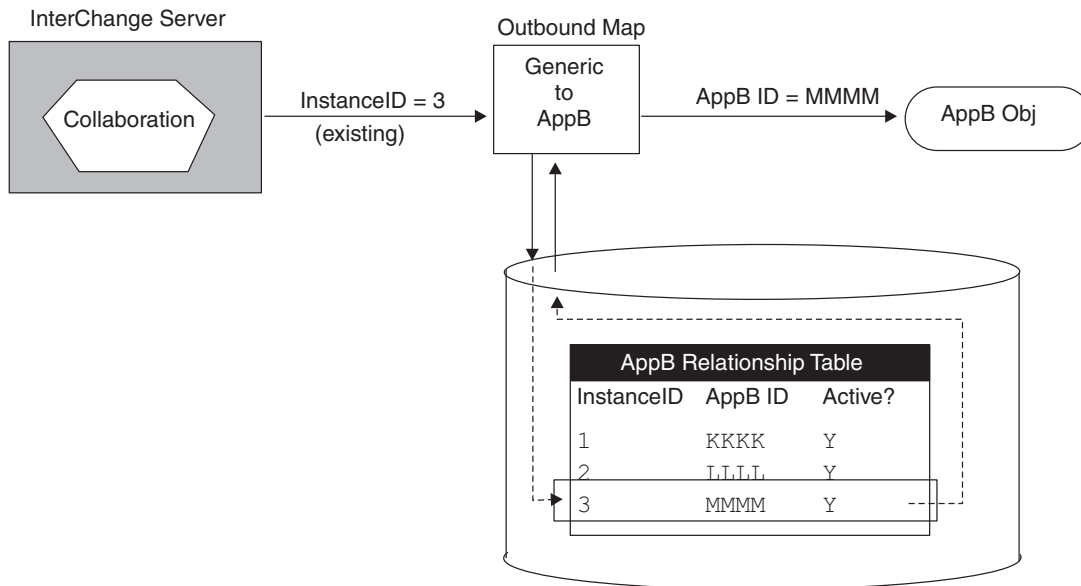


Figure 127. SERVICE\_CALL\_REQUEST with an update, delete, or retrieve verb

## SERVICE\_CALL\_RESPONSE calling context

When the calling context is `SERVICE_CALL_RESPONSE`, the map that is being called is an inbound map; that is, it transforms an application-specific business object to a generic business object. The inbound map receives an application-specific business object as input and returns a generic business object as output. The `SERVICE_CALL_RESPONSE` calling context is important for the Create verb, to indicate that the destination application was able to create a unique value for the new entity and the connector has returned an application-specific business object.

The task for the Cross-Reference transformation rule is to maintain an application-specific business object's key value in the relationship table for an existing relationship instance ID. For the `SERVICE_CALL_RESPONSE` calling context, the Java code generated by the Cross-Reference transformation takes the following actions:

1. Determines whether the generic business object is null:
  - For the Update, Delete, and Retrieve verbs, the transformation throws the `RelationshipRuntimeException` if the generic business object is null.
  - For a Create verb, a null-valued generic business object is valid.
2. Locates the entry in the relationship table that matches the given application-specific business object's key value. Table 97 shows the action that the Java code generated by the Cross-Reference transformation takes on the relationship table based on the verb of the application-specific business object.

Table 97. Actions for the `SERVICE_CALL_RESPONSE` calling context

Verb of application-specific business object	Action performed by <code>maintainSimpleIdentityRelationship()</code>
Create	For the given application-specific key, insert into the relationship table the new relationship entry containing the application-specific business object's key value and its associated relationship instance ID. The method obtains the relationship instance ID from the original-request business object in the map execution context ( <code>cwExecCtx</code> ).
Delete	If an entry for this key value already exists, retrieve the existing one; do <i>not</i> add another one to the table. <ol style="list-style-type: none"><li>1. Retrieve the relationship entry from the relationship table for the given application-specific business object's key value.</li><li>2. Mark the relationship entry as "deactive."</li></ol>
Update	Retrieve the relationship entry from the relationship table for the given application-specific business object's key value.
Retrieve	Retrieve the relationship entry from the relationship table for the given application-specific business object's key value.

For an identity relationship that supports the transformation of an AppA application-specific business object to AppB application-specific business object, Figure 128 shows how the Java code generated by the Cross-Reference transformation accesses a relationship table associated with the AppB participant when a calling context is `SERVICE_CALL_RESPONSE` and the AppB application-specific business object's verb is Create.

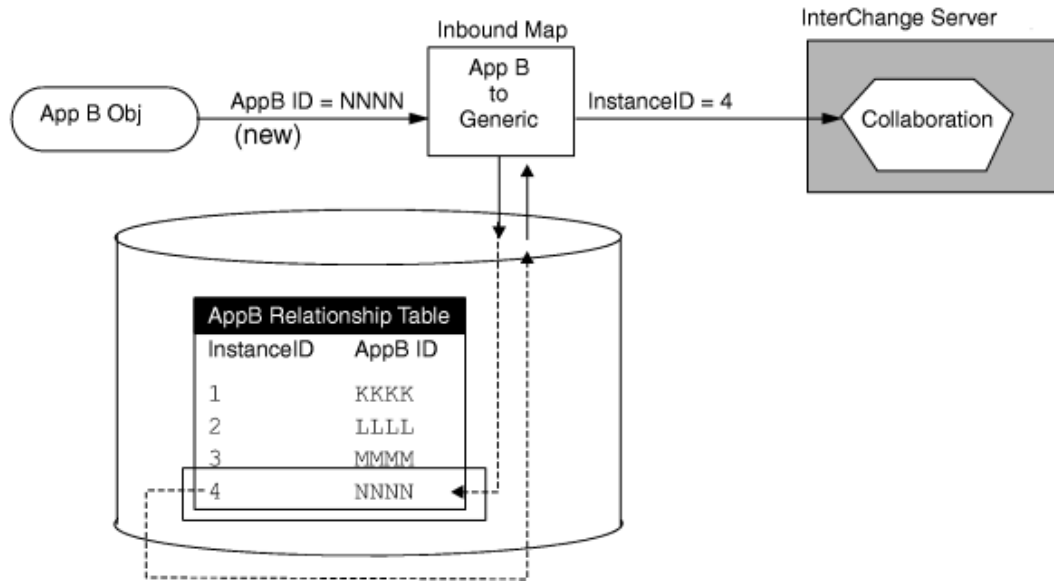


Figure 128. SERVICE\_CALL\_RESPONSE with the create verb

When the calling context is SERVICE\_CALL\_RESPONSE and the verb is Create, the inbound map has been invoked by the connector controller in response to the following actions:

- The connector has been notified that the application has inserted a new row. The connector sent this insert request to the application when it received the application-specific business object with a Create verb from the outbound map. This outbound map had a calling context of SERVICE\_CALL\_REQUEST. When the calling context was SERVICE\_CALL\_REQUEST, the Cross-Reference transformation could not write a new relationship instance to the relationship table because it did not yet have the application-specific key value that corresponded to the instance ID.
- The connector has generated a new application-specific business object based on the values in the new application-specific row and with a verb of Create. The connector sends this application-specific business object to InterChange Server where it is received by the connector controller.
- The connector controller has called the inbound map to convert the application-specific business object to a generic business object. The inbound map contains a Cross-Reference transformation to create an entry in the relationship table for the new application-specific key.

For a calling context of SERVICE\_CALL\_RESPONSE and an application-specific verb of Create, Figure 129 shows the write that the Java code generated by the Cross-Reference transformation makes to the relationship table.



Before Create			After Create		
AppB Relationship Table			AppB Relationship Table		
InstanceID	AppB ID	Active?	InstanceID	AppB ID	Active?
1	KKKK	Y	1	KKKK	Y
2	LLLL	Y	2	LLLL	Y
3	MMMM	Y	3	MMMM	Y
			4	NNNN	Y

"Inserted" Row

Figure 129. The write to the relationship table for a create verb

The Cross-Reference transformation must associate the new AppB application-specific key with its equivalent value in the AppA application. For the `EVENT_DELIVERY` or `ACCESS_REQUEST` calling context, the Cross-Reference transformation could just generate a new relationship instance ID. However, for `SERVICE_CALL_RESPONSE`, the Cross-Reference transformation cannot just generate a new instance ID. Instead, it must assign the same relationship instance ID to the AppB key value as it has already assigned to the AppA key value. The method obtains the instance ID associated with the AppA key value from the original-request business object, which is part of the map execution context.

In Figure 129, the Java code generated by the Cross-Reference transformation takes the following steps for the `SERVICE_CALL_RESPONSE` calling context and the Create verb:

- Obtains the instance ID of 4 from the original-request business object in map execution context.
- Creates a new entry in the AppB relationship table for this instance ID (4) and the new application-specific key (NNNN).

When the map executions with both the `EVENT_DELIVERY` (or `ACCESS_REQUEST`) and `SERVICE_CALL_RESPONSE` calling contexts (and a Create verb) are complete, the relationship tables for AppA and AppB use common relationship instance IDs to associate their keys, as Figure 130 shows.

AppA Relationship Table			AppB Relationship Table		
InstanceID	AppA ID	Active?	InstanceID	AppB ID	Active?
1	AAA	Y	1	KKKK	Y
2	BBB	Y	2	LLLL	Y
3	CCC	Y	3	MMMM	Y
4	DDD	Y	4	NNNN	Y

Relationship Instance

Figure 130. Creating the relationship instance

For the Update and Delete verbs (and Retrieve, if the instance ID already exists in the relationship table), the Cross-Reference transformation just retrieves the relationship instance ID from the relationship table. For a calling context of `SERVICE_CALL_RESPONSE` and an application-specific verb of Delete, the Cross-Reference transformation must take an additional step to deactivate the relationship instance, as Figure 131 shows.

Before Delete			After Delete		
AppB Relationship Table			AppB Relationship Table		
InstanceID	AppB ID	Active?	InstanceID	AppB ID	Active?
1	KKKK	Y	1	KKKK	Y
2	LLLL	Y	2	LLLL	N
3	MMMM	Y	3	MMMM	Y
4	NNNN	Y	4	NNNN	Y

"Deleted" Row

Figure 131. The write to the relationship table for `SERVICE_CALL_RESPONSE` and a delete verb

### SERVICE\_CALL\_FAILURE calling context

When the calling context is `SERVICE_CALL_FAILURE`, the map that is being called is an inbound map; that is it transforms an application-specific business object to a generic business object. For `SERVICE_CALL_FAILURE`, the inbound map receives a null application-specific business object as input and returns a generic business object as output. The `SERVICE_CALL_FAILURE` calling context is important for the Create verb; it indicates that the destination application was unable to create a unique value for the new entity and therefore the connector was unable to return an application-specific business object. The task for the Cross-Reference transformation is the same for all verbs, as Table 98 shows.

Table 98. Actions for the `SERVICE_CALL_FAILURE` calling context

Verb of Application-Specific business object	Action Performed by <code>maintainSimpleIdentityRelationship()</code>
Create Delete Update Retrieve	<ol style="list-style-type: none"> <li>1. Obtain the key value (relationship instance ID) from the generic business object. This generic business object is in the map execution context.</li> <li>2. Copy the retrieved instance ID into the generic business object.</li> </ol>

### ACCESS\_RESPONSE calling context

When the calling context is `ACCESS_RESPONSE`, the map that is being called is an outbound map as a result of a call-triggered flow. It transforms a generic business object to an application-specific business object. The outbound map receives a generic business object as input and returns an application-specific business object as output. Therefore, the task for the Cross-Reference transformation is the same for all verbs, as Table 99 shows.

Table 99. Actions for the `ACCESS_RESPONSE` calling context

Verb of generic business object	Action Performed by <code>maintainSimpleIdentityRelationship()</code>
Create Delete Update Retrieve	<ol style="list-style-type: none"> <li>1. Obtain the key value (relationship instance ID) from the generic business object. This generic business object is in the map execution context.</li> <li>2. Convert the relationship instance ID to an integer value. If this conversion fails, throw an exception.</li> <li>3. Copy the key values from the original-request business object into the application-specific business object.</li> </ol>

Because the original-request business object for `ACCESS_RESPONSE` is the application-specific business object, the Cross-Reference transformation automatically obtains this key value from the original-request business object in the map execution context (`cxExecCtx`).

The Cross-Reference transformation can perform the tasks in Table 99 as long as it has access to the original-request business object. However, in some cases, it might not have access to this business object. For example, if the Cross-Reference transformation is processing a child object that did not exist in the primary request, the method tries to retrieve that child object's relationship instance ID. If the method cannot find the relationship instance, it just populates the keys of this child object with the `CxIgnore` value.

## Defining transformation rules for a simple identity relationship

For information on defining a Cross-Reference relationship, see “Cross-referencing identity relationships” on page 47.

### Coding a child-level simple identity relationship

If child business objects have a unique key attribute, you can relate these child business objects in a simple identity relationship.

The following sections describe the steps for coding this simple identity relationship:

- “Steps for creating the child relationship definition” on page 273
- “Steps for customizing the parent map” on page 273
- “Steps for customizing the submap” on page 274

**Steps for creating the child relationship definition:** To create a relationship definition for a simple identity relationship between child business objects, perform the following steps:

1. Create a participant definition whose participant type is the child business object.
2. Set the participant attribute to the key attribute of the child business object.  
**Tip:** Expand the child business object and select the key attribute.
3. Repeat steps 1 and 2 for each of the participants. As with all simple identity relationships, this relationship contains one participant for the generic business object and at least one participant for an application-specific business object. Each participant contains a single attribute: the key of the business object.

**Steps for customizing the parent map:** In the map for the parent business object (the main map), add the mapping code to the attribute that contains the child business object. In Activity Editor for this attribute, perform the following steps to code a simple identity relationship:

1. If you created a submap for the child object, call this submap from the child attribute of the main map. Usually mapping transformations for a child object are done within a submap, especially if the child object has multiple cardinality.
2. Use the General/APIs/Identity Relationship/Maintain Child Verb function block to set the source child objects' verbs for you.

The last parameter of the General/APIs/Identity Relationship/Maintain Child Verb function block is a boolean flag to indicate whether the child objects are participating in a composite relationship. Make sure you pass a value of `false` as the last argument to the General/APIs/Identity Relationship/Maintain Child Verb function block because this child object participates in a simple, not a

composite identity relationship. If the child object has a submap, call the General/APIs/Identity Relationship/Maintain Child Verb function block *before* the call to the submap. For more information, see “Setting the source child verb” on page 287.

**Note:** If the key attribute of the parent business object also participates in a simple identity relationship, define a Cross-Reference transformation in the main map, as described in “Cross-referencing identity relationships” on page 47.

**Steps for customizing the submap:** In the submap, perform the following steps:

1. Define a Move or Set Value transformation for the child business object.
2. Define a Cross-Reference transformation for the child business object and specify the relationship name and participant. For more information, see “Cross-referencing identity relationships” on page 47.

---

## Using composite identity relationships

An identity relationship establishes an association between business objects or other data on a *one-to-one* basis. A composite identity relationship relates two business objects through a composite key attribute.

The following sections describe the steps for working with composite identity relationships:

- “Creating composite identity relationship definitions” on page 274
- “Determining the relationship action” on page 275
- “Customizing map rules for a composite identity relationship” on page 277

### Creating composite identity relationship definitions

Identity relationship definitions differ from lookup relationship definitions in that the participant types are business objects, *not* of the type Data (the first selection in the participant types list). As with a simple identity relationship:

- The composite identity relationship consists of the generic business object and at least one application-specific business object.
- The participant type is a business object for *all* participants.

However, for a composite identity relationship, the participant attribute for every participant is a composite key. This composite key usually consists of a unique key from a parent business object and a nonunique key from a child business object.

#### Steps for creating composite identity relationship definitions

To create a relationship definition for a composite identity relationship, perform the following steps:

1. Create a participant definition whose participant type is the parent business object.
2. Set the first participant attribute to the key of the parent business object.  
**Tip:** Expand the parent business object and select the key attribute.
3. Set the second participant attribute to the key of the child attribute.  
**Tip:** Expand the parent business object, then expand the child attribute within the parent. Select the key attribute from this child object.
4. Repeat steps 1-3 for each of the participants. As with all composite identity relationships, this relationship contains one participant for the generic business object and at least one participant for a application-specific business object.

Each participant consists of two attributes: the key of the parent business object and the key of the child business object (from the attribute within the parent business object).

**Restriction:** To manage composite relationships, the server creates internal tables. A table is created for each role in the relationship. A unique *index* is then created on these tables across all *key attributes* of the relationship. (In other words, the columns which correspond to the key attributes of the relationship are the participants of the index.) The column sizes of the internal tables have a direct relation to the attributes of the relationship and are determined by the value of the MaxLength attribute for the relationship.

Databases typically have restrictions on the size of the indexes that can be created. For instance, DB2 has an index limitation of 1024 bytes with the default page size. Thus, depending on the MaxLength attribute of a relationship and the number of attributes in a relationship, you could run into an index size restriction while creating composite relationships.

**Important:**

- You must ensure that appropriate MaxLength values are set in the repository file for all *key attributes* of a relationship, such that the total index would never exceed the index size limitations of the underlying DBMS.

If the MaxLength attribute for type String is not specified, the default is nvarchar(255) in the SQLServer. Thus, if a relationship has *N* Keys, all of type String and the default MaxLength attribute of 255 bytes, the index size would be  $((N*255)*2) + 16$  bytes. You can see that you would exceed the SQLServer 7 limit of 900 bytes quite easily when *N* takes values of  $\geq 2$  for the default MaxLength value of 255 bytes for type String.

- Remember, too, that even when some DBMS'es support large indexes, it comes at the cost of performance; hence, it is always a good idea to keep index sizes to the minimum.

For more information on how to create a relationship definition for a composite identity relationship, see "Defining identity relationships" on page 244.

## Determining the relationship action

Table 100 shows the activity function blocks that the Mapping API provides to maintain a composite identity relationship from the child attribute of the parent source business object. The actions that these methods take depends on the source object's verb and the calling context.

*Table 100. Maintaining a composite identity relationship from the child attribute*

Function block	Description
General/APIs/Identity Relationship/Maintain Child Verb	Set source child verb correctly
General/APIs/Identity Relationship/Maintain Composite Relationship	Perform appropriate action on the relationship tables

### Actions of General/APIs/Identity Relationship/Maintain Composite Relationship

The Maintain Composite Relationship function block will generate Java code that calls the mapping API `maintainCompositeRelationship()`, which will manage relationship tables for a composite identity relationship. This method ensures that the relationship instances contain the associated application-specific key values for

each relationship instance ID. This method automatically handles all of the basic adding and deleting of participants and relationship instances for a composite identity relationship.

The actions that `maintainCompositeRelationship()` takes are based on the value of the business object's verb and the calling context. The method iterates through the child objects of a specified participant, calling the `maintainSimpleIdentityRelationship()` on each one to correctly set the child key value. As with `maintainSimpleIdentityRelationship()`, the action that `maintainCompositeRelationship()` takes is based on the following information:

- The calling context: `EVENT_DELIVERY`, `ACCESS_REQUEST`, `SERVICE_CALL_REQUEST`, `SERVICE_CALL_RESPONSE`, `SERVICE_CALL_FAILURE`, and `ACCESS_RESPONSE`
- The verb of the source business object: `Create`, `Update`, `Delete`, or `Retrieve`

For information on the actions that `maintainSimpleIdentityRelationship()` takes, see "Accessing identity relationship tables" on page 263.

The `maintainCompositeRelationship()` method deals *only* with composite keys that extend to only two nested levels. In other words, the method cannot handle the case where the child object's composite key depends on values in its grandparent objects.

**Example:** If A is the top-level business object, B is the child of A, and C is the child of B, the two methods will *not* support the participant definitions for the child object C that are as follows:

- The participant type is A and the attributes are:
  - key attribute of A: ID
  - key attribute of B: `B[0].ID`
  - key attribute of C: `B[0].C[0].ID`
- The participant type is A and the attributes are:
  - key attribute of A: ID
  - key attribute of C: `B[0].C[0].ID`

To access a grandchild object, these methods only support the participant definitions that are as follows:

- The participant type is B and the attributes are:
  - key attribute of B: ID
  - key attribute of C: `C[0].ID`
- The participant type is B and the attributes are:
  - key attribute of B: ID
  - first key attribute of C: `C[0].ID1`
  - second key attribute of C: `C[0].ID2`

### **Actions of General/APIs/Identity Relationship/Maintain Child Verb**

The Maintain Child Verb function block will generate Java code that calls the mapping API `maintainChildVerb()`, which will maintain the verb of the child objects in the destination business object. It can handle child objects whose key attributes are part of a composite identity relationship. When you call `maintainChildVerb()` as part of a composite relationship, make sure that its last parameter has a value of `true`. This method ensures that the verb settings are appropriate given the verb in the parent source object and the calling context. For more information on the actions of `maintainChildVerb()`, see "Setting the source child verb" on page 287.

## Customizing map rules for a composite identity relationship

Once you have created the relationship definition and participant definitions for the composite identity relationship, you can customize the map to maintain the composite identity relationship. A composite identity relationship manages a composite key. Therefore, managing this kind of relationship involves managing both parts of the composite key. To code a composite identity relationship, you need to customize the mapping transformation rules for both the parent and child business objects, as Table 101 shows.

Table 101. Activity function blocks for a composite identity relationship

Map involved	Business object involved	Attribute	Activity function blocks
Main	Parent business object	Top-level business object	Use a Cross-Reference transformation rule
		Child attribute (child business object)	General/APIs/Identity Relationship/Maintain Composite Relationship General/APIs/Identity Relationship/Maintain Child Verb General/APIs/Identity Relationship/Update My Children (optional)
Submap	Child business object	Key attribute (nonunique key)	Define a Move or Set Value transformation for the verb.

If child business objects have a nonunique key attribute, you can relate these child business objects in a composite identity relationship.

The following sections describe the steps for customizing this composite identity relationship:

- “Steps for customizing the main map”
- “Customizing the submap” on page 281
- “Managing child instances” on page 282

### Steps for customizing the main map

In the map for the parent business object (the main map), add the mapping code to the parent attributes:

1. Map the verb of the top-level business object by defining a Move or Set Value transformation rule.
2. Define a Cross-Reference transformation between the top-level business objects.
3. Define a Custom transformation for the child attribute and use the General/APIs/Identity Relationship/Maintain Composite Relationship function block in Activity Editor.

### Steps for coding the child attribute

The child attribute of the parent object contains the child business object. This child object is usually a multiple cardinality business object. It contains a key attribute whose value identifies the child. However, this key value is not required to be unique. Therefore, it does not uniquely identify one child object among those for the same parent nor is it sufficient to identify the child object among child objects for all instances of the parent object.

To identify such a child object uniquely, the relationship uses a composite key. In the composite key, the parent key uniquely identifies the parent object. The combination of parent key and child key uniquely identifies the child object. In the map for the parent business object (the main map), add the mapping code to the



attribute that contains the child business object. In Activity Editor for this attribute, perform the following steps to code a composite identity relationship:

1. Define a Submap transformation for the child business object attribute of the main map. Usually mapping transformations for a child object are done within a submap, especially if the child object has multiple cardinality.
2. In the main map, define a Custom transformation rule for the child verb and use the General/APIs/Identity Relationship/Maintain Child Verb function block to maintain the child business object's verb.

The last input parameter of the General/APIs/Identity Relationship/Maintain Child Verb function block is a boolean flag to indicate whether the child objects are participating in a composite relationship. Make sure you pass a value of true as the last argument to `maintainChildVerb()` because this child object participates in a composite, not a simple identity relationship. Make sure you call `maintainChildVerb()` before the code that calls the submap. For more information, see "Setting the source child verb" on page 287.

3. To maintain this composite key for the parent source object, customize the mapping rule to use the General/APIs/Identity Relationship/Maintain Composite Relationship function block.
4. To maintain the relationship tables in the case where a parent object has an Update verb caused by child objects being deleted, customize the mapping rule to use the General/APIs/Identity Relationship/Update My Children function block.

**Tip:** Make sure the transformation rule that contains the Update My Children function block has an execution order after the transformation rule that contains the Maintain Composite Relationship function block.

## Example of customizing the map for a Composite Identity Relationship

The following example describes how the map can be customized for a Composite Identity Relationship.

1. In the main map, define a Custom transformation rule between the child business object's verbs. Use the General/APIs/Identity Relationship/Maintain Child Verb function block in the customized activity to maintain the verb for the child business objects.

The goal of this custom activity is to use the `maintainChildVerb()` API to set the child business object verb based on the map execution context and the verb of the parent business object. Figure 132 on page 279 shows this custom activity.

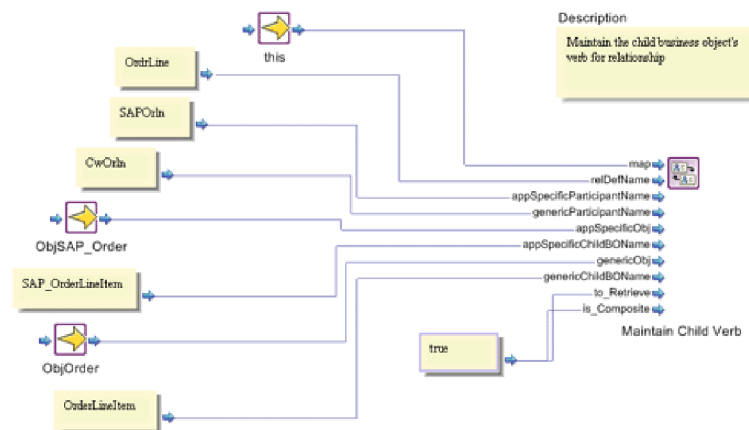




Figure 132. Using the Maintain Child Verb function block

2. If necessary, define a Submap transformation rule between the child business object to perform any mapping necessary in the child level.
3. Define a Custom transformation rule between the top-level business objects. Use the General/APIs/Identity Relationship/Maintain Composite Relationship function block in the customized activity to maintain the composite identity relationship for this map.

The goal of this custom activity is to use the `maintainCompositeRelationship()` API to maintain a composite identity relationship within the map. Figure 133 shows this custom activity.

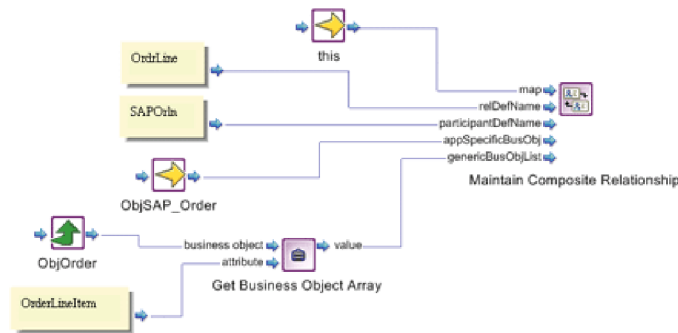


Figure 133. Using the Maintain Composite Relationship function block

4. Define a Custom transformation rule mapping from the source top-level business object to the destination child business object attribute. Use the General/APIs/Identity Relationship/Update My Children function block in the customized activity to maintain the child instances in the relationship.

The goal of this custom activity is to use the `updateMyChildren()` API to add or delete child instances in the specified parent/child relationship of the identity relationship. Figure 134 shows this custom activity.

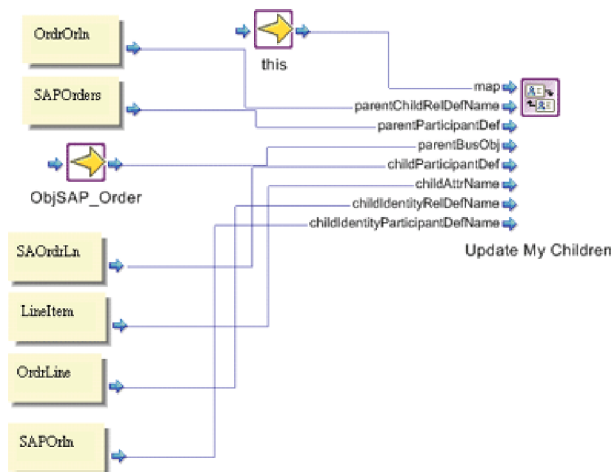


Figure 134. Using the Update My Children function block

## Example of coding the child attribute

Here is a sample of how the code in the child attribute of the parent map might look. This code fragment would exist in the Order Line Item attribute of an SAP Order business object. It uses `maintainChildVerb()` to set the child object verbs, then calls a submap (`Sub_SaOrderLineItem_to_CwOrderLineItem`) in a for loop to handle mapping of the Order line items child object:

```
{
BusObjArray srcCollection_For_ObjSAP_Order_SAP_OrderLineItem =
    ObjSAP_Order.getBusObjArray("SAP_OrderLineItem");

//
// LOOP ONLY ON NON-EMPTY ARRAYS
// -----
//
// Perform the loop only if the source array is non-empty.
//
if ((srcCollection_For_ObjSAP_Order_SAP_OrderLineItem != null) &&
    (srcCollection_For_ObjSAP_Order_SAP_OrderLineItem.size() > 0))
{
    int currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem;
    int lastInputIndex_For_ObjSAP_Order_SAP_OrderLineItem =
        srcCollection_For_ObjSAP_Order_SAP_OrderLineItem.getLastIndex();

    // ----
    IdentityRelationship.maintainChildVerb(
        "OrdrLine",
        "SAPOrln",
        "CWOOrln",
        ObjSAP_Order,
        "SAP_OrderLineItem",
        ObjOrder,
        "OrderLineItem",
        cwExecCtx,
        true,
        true);

    // ----
    for (currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem = 0;
         currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem <=
             lastInputIndex_For_ObjSAP_Order_SAP_OrderLineItem;
         currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem++)
    {
        BusObj currentBusObj_For_ObjSAP_Order_SAP_OrderLineItem =
            (BusObj) (srcCollection_For_ObjSAP_Order_SAP_OrderLineItem.elementAt(
                currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem));

        //
        // INVOKE MAP ON VALID OBJECTS
        // -----
        //
        // Invoke the map only on those children objects that meet
        // certain criteria.
        //
        if (currentBusObj_For_ObjSAP_Order_SAP_OrderLineItem != null)
        {
            BusObj[] _cw_inObjs = new BusObj[2];
            _cw_inObjs[0] =
                currentBusObj_For_ObjSAP_Order_SAP_OrderLineItem;
            _cw_inObjs[1] = ObjSAP_Order;
            LogInfo ("*** Inside SAPCW header, verb is: " +
                (_cw_inObjs[0].getVerb()));

            try
            {
```

```

        BusObj[] _cw_outObjs = DtpMapService.runMap(
            "Sub_SaOrderLineItem_to_CwOrderLineItem",
            "CwMap",
            _cw_inObjs,
            cwExecCtx);
        _cw_outObjs[0].setVerb(_cw_inObjs[0].getVerb());
        ObjOrder.setWithCreate("OrderLineItem", _cw_outObjs[0]);
    }

    catch (MapNotFoundException me)
    {
        logError(5502,
            " Sub_SaOrderLineItem_to_CwOrderLineItem ");
        throw new MapFailureException ("Submap not found");
    }
}

// Start of the child relationship code
BusObjArray temp = (BusObjArray)ObjOrder.get("OrderLineItem");
try
{
    IdentityRelationship.maintainCompositeRelationship(
        "OrdrLine",
        "SAPOrLn",
        ObjSAP_Order,
        temp,
        cwExecCtx);
}

catch RelationshipRuntimeException re
{
    logError(re.toString());
}

// This call to updateMyChildren() assumes the existence of the
// OrdrOrLn parent/child relationship between the SAP_Order
// (parent) and SAP_OrderItem (child)
IdentityRelationship.updateMyChildren(
    "OrdrOrLn",
    "SAOrders",
    ObjSAP_Order,
    "SAOrdrLn",
    "LineItem",
    "OrdrLine",
    "SAPOrLn",
    cwExecCtx);

// End of the child relationship code
}
}

```

## Customizing the submap

In the map for the child business object (the submap), add the mapping code to the the key attribute of the child object. The only code you need to add is a call to the `setVerb()` method to set the child object's verb to the parent object's verb. For more information, see "Setting the destination business object verb" on page 36.

**Note:** When the child object primary key requires the `maintainCompositeRelationship()` method, make the call in the parent map, right after the end of the for loop for calling the submap. In the submap, the code for the destination object's primary key should contain the following line:

```
// maintainCompositeRelationship()  
is called in the parent map.
```

---

## Managing child instances

Activity Editor provides the function blocks in Table 102 to manage child object instances that belong to a parent in an identity relationship.

Table 102. Function blocks for Managing Child Instances

Function block	Description
General/APIs/Identity Relationship/Add My Children	Adds child relationship instances to parent/child relationship tables
General/APIs/Identity Relationship/Delete My Children	Deletes child relationship instances to parent/child relationship tables
General/APIs/Identity Relationship/Update My Children	Deletes or adds child relationship instances from parent/child relationship tables.

**Note:** The most common use of the function blocks in Table 102 is to maintain child business objects in custom relationships involving composite identity relationships.

The function blocks in Table 102 assume that the parent business object being passed is an after-image; that is, the image of the business object *after* the verb operation has taken place. For example, if a business object has an Update verb with the update caused by the addition of new child objects, these new child objects already exist in the business object. Similarly, if a business object has an Update verb with the update caused by the deletion of child objects, the business object already has these child objects deleted.

The following sections describe the steps for managing child instances:

- “Creating the parent/child relationship definition”
- “Handling updates to the parent business object” on page 283

### Creating the parent/child relationship definition

A *parent/child relationship* is a 1-to-many relationship between parent (1) and child (many) business objects. A parent/child relationship involves the following participants:

- A participant containing the key attribute of that parent business object
- A participant containing the key of the child business object

The relationship tables for a parent/child relationship enable the function blocks in Table 102 to track the child business objects associated with a particular parent business object.

#### Steps for creating the parent/child relationship definition

To create a relationship definition for a parent/child relationship, perform the following steps in Relationship Designer:

1. Create a participant definition whose participant type is the parent business object.
2. Set the participant attribute to the key of the parent business object:  
Expand the parent business object and select the key attribute.
3. Create a participant definition whose participant type is the child business object.

4. Set the participant attribute to the key of the child attribute:  
Expand the child business object (not the child attribute with the parent object), and select the key attribute from this child object.

**Note:** The parent-child relationship needs to be maintained only if the child object does *not* have a unique key; that is, the child object *only* exists within the context of its parent.

For more information, see “Defining identity relationships” on page 244.

## Handling updates to the parent business object

This section provides information to ensure that child objects that participate in a composite identity relationship are correctly managed during an Update:

- “Comparing the before- and after-images”
- “Tips on using Update My Children” on page 284

### Comparing the before- and after-images

The Update My Children function block updates the relationship tables for a parent/child relationship. A parent/child relationship is needed to help determine whether child objects have been added to or deleted from a parent business object.

For a given parent business object, this method makes sure that the following images of the business object match:

- The before-image information is contained in the relationship tables for the parent/child relationship.
- The after-image is contained in the parent business object.

For the map to detect that a child business object has been deleted, it must determine how many instances of the child object of this type that the parent business object had before the Update (the before-image) and compare that to what the parent object presently has (the after-image). The map can use the Update My Children function block to make this comparison and find out what has been deleted or added.

When Update My Children compares the before- and after-images, it can determine whether to remove the associated relationship instances from the relationship tables for any child object that is *not* present in the parent business object. The method removes relationship instances from the following relationship tables:

- The relationship table for the child participant in the parent/child relationship
- The relationship table for the participant in the composite identity relationship that contains the parent and child objects

**Note:** Although Update My Children can also add instances to the relationship table for any child object that *is* present in the parent business object (but not in the child relationship table), it does not need to do this when called in the context of a composite identity relationship. All new child objects for the parent object have already been added to the relationship tables by the Maintain Composite Relationship function block. For more information, see “Actions of General/APIs/Identity Relationship/Maintain Composite Relationship” on page 275.

## Tips on using Update My Children

When you use the Update My Children function block to maintain relationship tables for a child object involved in a composite identity relationship, keep the following tips in mind:

- Make sure you use the Update My Children function block *after* the Maintain Composite Relationship function block and that you have set the appropriate verbs on the child business objects.
- The Update My Children function block is only needed to track child objects involved in composite relationships.

You do *not* need to use the Update My Children function block to track child objects involved in a simple identity relationship. For more information, see “Coding a child-level simple identity relationship” on page 273.

- The Update My Children function block (as with the Maintain Composite Relationship function block) deals only with composite keys that extend to only two nested levels: the parent and its immediate children.

In other words, the method cannot handle the case where the grandchild object’s composite key depends on values in its grandparent objects. For example, if A is the top-level business object, B is the child of A, and C is the child of B, the two methods will not support the participant definitions for the child object C that are as follows:

- The participant type is A and the attributes are:

- key attribute of A: ID
- key attribute of B: B[0].ID
- key attribute of C: B[0].C[0].ID

- The participant type is A and the attributes are:

- key attribute of A: ID
- key attribute of C: B[0].C[0].ID

To access a grandchild object, these methods only support the participant definitions that are as follows:

- The participant type is B and the attributes are:

- key attribute of B: ID
- key attribute of C: C[0].ID

- The participant type is B and the attributes are:

- key attribute of B: ID
- first key attribute of C: C[0].ID1
- second key attribute of C: C[0].ID2

- The Update My Children function block manages the parent/child relationship tables for the EVENT\_DELIVERY and SERVICE\_CALL\_RESPONSE calling contexts *only*. Execution of the Update My Children function block with a calling context of SERVICE\_CALL\_REQUEST or ACCESS\_RESPONSE does *not* produce any changes to these relationship tables.
- The Update My Children function block can also be used when the child business object has a unique ID; that is, the child object participates in a simple identity relationship. In this case, you must still define the parent/child relationship (see “Creating the parent/child relationship definition” on page 282).

---

## Setting the verb

This section provides information on setting the verb of a business object participating in a map:

- “Conditionally setting the destination verb”
- “Setting the source child verb” on page 287

For information on setting the verb of the destination business object, see “Setting the destination business object verb” on page 36.

### Conditionally setting the destination verb

Usually, you just set the destination verb to the value of the source verb by defining a Move transformation. (For more information on this action, see “Setting the destination business object verb” on page 36.) However, sometimes the source application sets the business object verb in an unusual manner; for example, the verb is set to Update even though the event is new. As another example, the verb is always set to Retrieve. In the situations like these, the map must reset the destination verb to the one that corresponds to the actual event.

If the source business object’s key participates in a relationship, the map can perform a *static lookup* in the relationship table to determine if the source business object exists. The map can then set the destination verb to either Update or Create based on whether the corresponding entry is found in the table. You perform this static lookup in much the same way as accessing a lookup relationship. Table 103 shows the function block to use for each kind of static lookup.

Table 103. Checking for Existence of the source business object

Type of source business object	Map type	Function block
Application-specific	Inbound	General/APIs/Relationship/ Retrieve Instances
Generic	Outbound	General/APIs/Relationship/ Retrieve Participants

### Example of steps for customizing the inbound map

The following example shows how an inbound map can conditionally set the destination verb based on the result of a lookup:

1. In the map, define a Custom transformation between the source business object and the destination verb.
2. In the activity of this Custom Transformation, perform the following steps. The goal of this activity is to identify the number of instances in the participant of the relationship. If there are no participant instances in the relationship, the destination business object verb should be Create; otherwise, the verb should be Update.

- a. Define the activity, as shown in Figure 135, to identify the number of instances in the relationship participant.

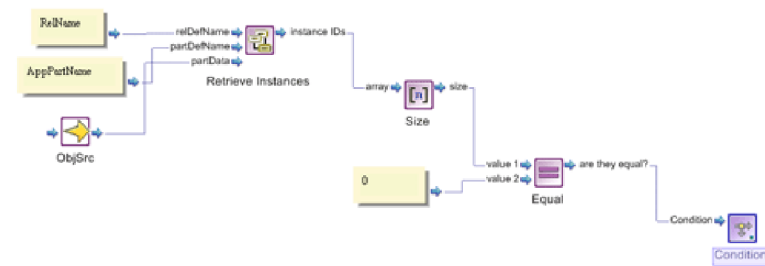


Figure 135. Identifying the number of instances in the relationship participant

- b. Double-click the Condition function block in the canvas to open it. Select True Action to define the action to take when the condition is true. Define the True Action as shown in Figure 136.

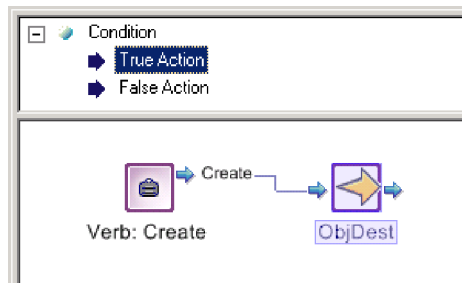


Figure 136. Defining the True Action

- c. Select the False Action to define the action to take when the number of participant instances is not zero. Define the False Action as shown in Figure 137.

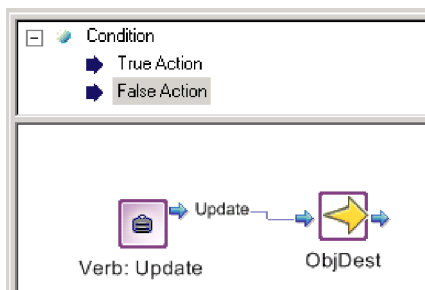


Figure 137. Defining the False Action

### Example of steps for customizing the outbound map

You can use similar steps in the outbound map to perform a static lookup based on the primary key of the generic object. To do that, you need to replace the function block General/APIs Relationship/Retrieve Instances with the function block General/APIs Relationship/Retrieve Participants. Here are the steps:

1. In the map, define a Custom transformation between the key attribute of the source business object and the destination verb.



2. In the activity of this Custom transformation, perform the following steps. The goal of this activity is to identify the number of participants of the relationship. If there are no participant instances in the relationship, the destination business object verb should be Create; otherwise, the verb should be Update.
  - a. Define the activity, as shown in Figure 138, to identify the number of participants in the relationship.

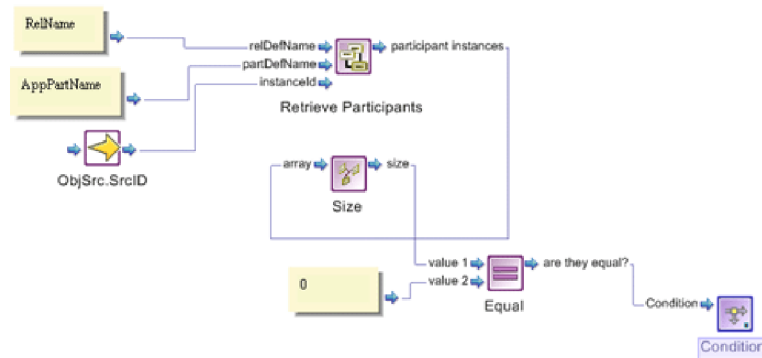


Figure 138. Identifying the number of participants in the relationship

- b. Follow steps 2b and 2c, described in “Example of steps for customizing the inbound map” on page 285.

## Setting the source child verb

When a parent source business object has child business objects, the value of the source child verb is usually the same as that of the parent verb. Therefore, you set the source child object’s verb by defining a Move transformation from the parent verb to the child verb. However, if the parent object’s verb is Update, the update could be the result of any of the modifications shown in Table 104.

Table 104. Updating a parent business object

Update task	Verb in child object
Modifying some non-child attribute in the parent object	Update
Modifying some attribute in a child object	Update
Adding more child objects	Create
Deleting existing child objects	Delete

All of the modifications in Table 104 are represented by a verb of Update in the parent object. However, not all of these modifications represent an Update to the child object. The value of the source child verb depends on what action was taken on the parent verb. When the child object’s key participates in an identity relationship (composite or simple), the source child verb value depends not just on the parent verb but also on the calling context. In such cases, use the Maintain Child Verb function block to handle the setting of the verb of the source child object.

This section provides the following information about using the Maintain Child Verb function block to maintain a source child object verb:

- “Determining the child verb setting” on page 288
- “Tips for using the Maintain Child Verb function block” on page 289

## Determining the child verb setting

The Maintain Child Verb function block must ensure that the verb settings of the child objects in the source business object are appropriate, given the verb in the parent source object and the calling context. The actions that this method takes are based on the verb in the parent source object and the calling context.

**EVENT\_DELIVERY and ACCESS\_REQUEST calling contexts:** When the calling context is `EVENT_DELIVERY` or `ACCESS_REQUEST`, the map that is being called is an inbound map; that is, it transforms an application-specific business object to a generic business object. The inbound map receives an application-specific business object as input and returns a generic business object as output. For `EVENT_DELIVERY` (or `ACCESS_REQUEST`), there are no special cases to handle when setting the child verbs. Therefore, the `maintainChildVerb()` method just copies the parent verb to the child verb for all verb values, as Table 105 shows.

Table 105. Actions for the `EVENT_DELIVERY` and `ACCESS_REQUEST` calling contexts

Verb of generic business object	Action performed by the Maintain Child Verb function block
Create Delete Update Retrieve	Set the verbs of all child objects in the source object to the verb in the parent source object. This action overwrites any existing verb in the child object.

**SERVICE\_CALL\_REQUEST calling context:** When the calling context is `SERVICE_CALL_REQUEST`, the map that is being called is an outbound map; that is, it transforms a generic business object to an application-specific business object. The outbound map receives a generic business object as input and returns an application-specific business object as output. For `SERVICE_CALL_REQUEST`, the Java code generated by the Maintain Child Verb function block handles the special case for an Update verb: If the change to the parent object is the creation of new child objects, the Maintain Child Verb function block changes the verb to Create for any child objects that do not currently exist in the relationship tables, as Table 106 shows.

Table 106. Actions for `SERVICE_CALL_REQUEST` calling context

Verb of generic business object	Action performed by the Maintain Child Verb function block
Create Delete Retrieve Update	Set the verbs of all child objects in the source object to the verb in the parent source object. This action overwrites any existing verb in the child object. <ol style="list-style-type: none"><li>Retrieve the relationship instance from the child relationship table for the given generic business object's key value.</li><li>Set the verb of the child object based on the success of the table lookup:<ul style="list-style-type: none"><li>If a relationship instance for this child object exists, set the verb of the child object to Update.</li><li>If a relationship instance for this child object does <i>not</i> exist, set the verb of the child object to Create.</li></ul></li></ol>

**SERVICE\_CALL\_RESPONSE calling context:** When the calling context is `SERVICE_CALL_RESPONSE`, the map that is being called is an inbound map; that is it transforms an application-specific business object to a generic business object. The inbound map receives an application-specific business object as input and returns a generic business object as output.

The behavior of the Maintain Child Verb function block is determined by the second-to-last parameter of the method. This parameter is the boolean `toRetrieve`

flag, whose value indicates whether the application resets or preserves child objects' verbs when processing a collaboration request, as Table 107 shows.

Table 107. Connector behavior

Value of <code>to_Retrieve</code> flag	Connector behavior
true	Connector sets child object verbs to different value from what they had coming into the application.  For example, if a business object comes to the connector with a parent verb of Update and a child verb of Create, the connector might reset all child object verbs to their parent value after the application completes the operation. In this case, the child verb would be changed to Update.
false	Connector preserves child object verbs.  For example, if a business object comes to the connector with a parent verb of Update and a child verb of Create, the connector preserves all child object verbs. In this case, the child verb would still be Create.

**Note:** The Java code generated by the Maintain Child Verb function block uses the value of the `to_Retrieve` parameter *only* when it processes the `SERVICE_CALL_RESPONSE` calling context.

If the `to_Retrieve` argument is true, the Maintain Child Verb function block performs the tasks in Table 108.

Table 108. Actions for the `SERVICE_CALL_RESPONSE` calling context

Verb of generic business object	Action performed by the Maintain Child Verb function block
Create Delete Retrieve Update	Set the verbs of all child objects in the source object to the verb in the parent source object. This action overwrites any existing verb in the child object.  1. Look up each child object in the child relationship table. 2. Set the verb of the child object based on the success of the table lookup: <ul style="list-style-type: none"> <li>• If a relationship instance for this child object exists, set the verb of the child object to Update.</li> <li>• If a relationship instance for this child object does <i>not</i> exist, set the verb of the child object to Create.</li> </ul>

**Note:** If you are unsure of the behavior of your application, set the `to_Retrieve` argument to true. With a true flag value, performance might be affected because the Java code generated by the Maintain Child Verb function block might perform an unnecessary lookup. However, it is usually safer to have an unnecessary lookup than to have an incorrect verb setting in the child object.

### Tips for using the Maintain Child Verb function block

The Maintain Child Verb function block maintains the verb of the child objects in the source business object. It can handle child objects that are part of a simple or a composite identity relationship. This function block must ensure that the verb settings are appropriate given the verb in the parent source object and the calling context.

Keep the following tips in mind when using the Maintain Child Verb function block:

- The second to last parameter in this method is the *to\_Retrieve* boolean flag, which indicates whether the application resets or preserves child objects' verbs. For more information on how to set the *to\_Retrieve* flag, see "SERVICE\_CALL\_RESPONSE calling context" on page 288.
- The last parameter in this method is the *is\_Composite* boolean flag, which indicates whether the child object is part of a simple or composite identity relationship.
 

The key attribute of a child business object can participate in either of the following kinds of identity relationship:

  - As a unique key in a simple identity relationship  
Set the value of the *is\_Composite* flag to false.
  - As a nonunique key of a composite key in a composite identity relationship; in this case, the other part of the composite key is the unique key in the parent business object.  
Set the value of the *is\_Composite* flag to true.
- Make sure you use the Maintain Child Verb function block in the child attribute of the source parent map, *before* calling the submap.
 

For multiple-cardinality child objects, use the Maintain Child Verb function block right *before* the start of the for loop. The method iterates through the child objects to set the child verbs correctly.

---

## Performing foreign key lookups

A *foreign key* is an attribute within one business object that contains the key value of another business object. This key value is considered "foreign" to the source business object because it identifies some other business object. To transform a foreign key in a source business object, you must access the relationship table associated with the business object that the foreign key references (the foreign relationship table). From this foreign relationship table, you can obtain the associated key value for the foreign key of the destination business object.

The Mapping API provides the methods in Table 109 to perform foreign key lookups.

*Table 109. Function blocks for foreign key lookups*

Function block	Description
General/APIs/Identity Relationship/Foreign Key Lookup	Performs a foreign key lookup, failing to find a relationship instance if the foreign key does not exist in the foreign relationship table.
General/APIs/Identity Relationship/Foreign Key Cross-Reference	Performs a foreign key lookup, adding a new relationship instance in the foreign relationship table if the foreign key does not exist.

## Using the Foreign Key Lookup function block

The Java code generated by the Foreign Key Lookup function block performs a lookup in a foreign relationship table for the foreign key of the source business object. This function block takes the following actions:

1. Verifies that the application-specific participant contains a single key, *not* a composite key.

Determines the participant type of the application-specific participant, which is the application-specific business object. In this business object, verifies that only one key attribute exists. If more than one key attribute exists, the Foreign Key

Lookup function block does not know which application-specific key attribute to populate with the application-specific equivalent of the generic business object's foreign key. Therefore, it throws the `RelationshipRuntimeException` exception.

2. Locates the relationship instance in the foreign relationship table that matches the value of the foreign key in the generic business object.
3. Obtains the application-specific key value from the retrieved relationship instance.
4. Copies the application-specific key value into the foreign key of the application-specific business object.

The Java code generated by the Foreign Key Lookup function block takes these actions on the foreign relationship table regardless of the verb in the source business object.

## Using the Foreign Key Cross-Reference function block

As with the Foreign Key Lookup function block, the Foreign Key Cross-Reference function block performs a lookup in a foreign relationship table based on the foreign key of the source business object. However, the Foreign Key Cross-Reference function block provides the additional functionality that it can add an entry to the foreign relationship table if the lookup fails. The following sections discuss the behavior of the Foreign Key Cross-Reference function block with each of the calling contexts.

### **EVENT\_DELIVERY, ACCESS\_REQUEST, and SERVICE\_CALL\_RESPONSE calling contexts**

When the calling context is `EVENT_DELIVERY`, `ACCESS_REQUEST`, or `SERVICE_CALL_RESPONSE`, the map that is being called is an inbound map; that is, it transforms an application-specific business object to a generic business object. The inbound map receives an application-specific business object as input and returns a generic business object as output. Therefore, the task for the Foreign Key Cross-Reference function block is to obtain from the foreign relationship table the generic key for a given application-specific key value.

For the `EVENT_DELIVERY`, `ACCESS_REQUEST`, and `SERVICE_CALL_RESPONSE` calling contexts, the Foreign Key Cross-Reference function block takes the following actions:

1. Verifies that the generic participant contains a single key, *not* a composite key. Determines the participant type of the generic participant, which is the generic business object. In this business object, verifies that only one key attribute exists. If more than one key attribute exists, the Foreign Key Cross-Reference function block does not know which generic key attribute to populate with the generic equivalent of the application-specific business object's foreign key. Therefore, it throws the `RelationshipRuntimeException` exception.
2. Locates the relationship instance in the foreign relationship table that matches the value of the foreign key in the application-specific business object. Table 110 shows the actions that the Foreign Key Cross-Reference function block takes on the foreign relationship table based on the verb of the application-specific business object.
3. Obtains the instance ID from the retrieved relationship instance.

4. Copies the instance ID into the foreign key of the generic business object.

Table 110. Actions for *EVENT\_DELIVERY*, *ACCESS\_REQUEST*, and *SERVICE\_CALL\_RESPONSE*

Verb of application-specific business object	Action performed by the Foreign Key Cross-Reference function block
Create	<p>For the <i>EVENT_DELIVERY</i> and <i>ACCESS_REQUEST</i> calling contexts, insert a new relationship entry into the foreign relationship table for the application-specific business object's key value.</p> <p>For the <i>SERVICE_CALL_RESPONSE</i> calling context, insert into the relationship table the new relationship entry containing the application-specific business object's key value and its associated relationship instance ID. The method obtains the relationship instance ID from the original-request business object in the map execution context (cwExecCtx). For more information on the behavior of the <i>SERVICE_CALL_RESPONSE</i>, see "SERVICE_CALL_RESPONSE calling context" on page 269.</p> <p>If an entry for this key value already exists, retrieve the existing one; do <i>not</i> add another one to the table.</p>
Update	<p>Retrieve the relationship entry from the foreign relationship table for the given application-specific business object's foreign key value.</p> <p>If an entry for this foreign key value does <i>not</i> exist, insert a new relationship instance into the foreign relationship table for the application-specific business object's foreign key value.</p>
Retrieve	<p>Retrieve the relationship entry from the foreign relationship table for the given application-specific business object's foreign key value</p>

Figure 139 shows how the Foreign Key Cross-Reference function block accesses the foreign relationship table (for App Obj C) when a calling context is *EVENT\_DELIVERY*, *ACCESS\_REQUEST*, or *SERVICE\_CALL\_RESPONSE* and the verb for the application-specific business object (App Obj A) is either Create or Update.

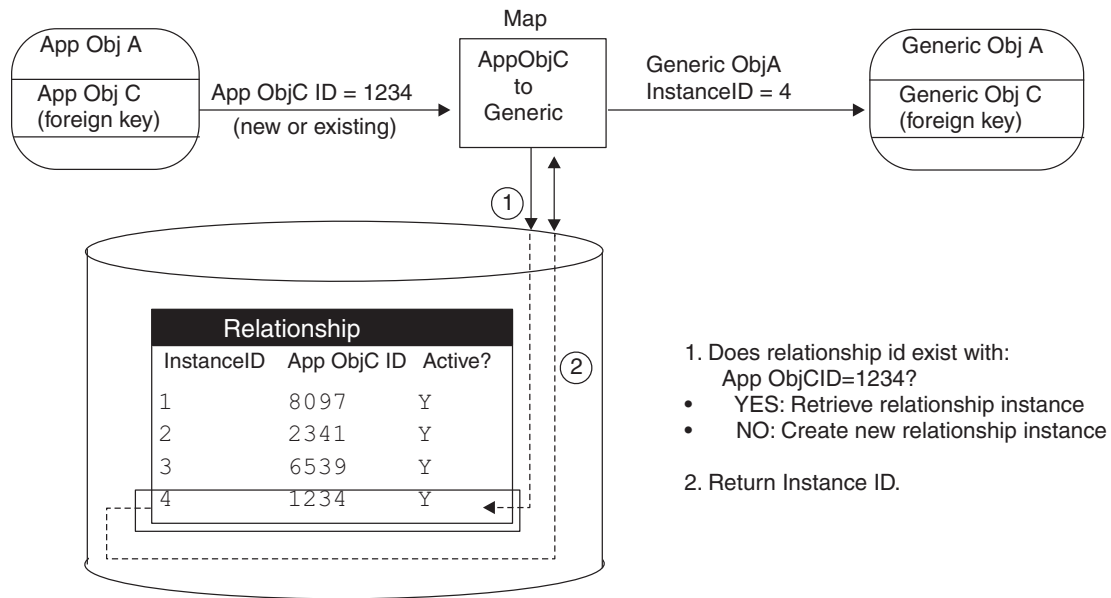


Figure 139. Foreign key lookup for a create or update verb

**Note:** The Foreign Key Cross-Reference function block only adds relationship instances to the foreign relationship table for inbound maps.

### **SERVICE\_CALL\_REQUEST calling context and Foreign Keys**

When the calling context is `SERVICE_CALL_REQUEST`, the map that is being called is an outbound map; that is, it transforms a generic business object to an application-specific business object. The outbound map receives a generic business object as input and returns an application-specific business object as output. For the `SERVICE_CALL_REQUEST` calling context, the Foreign Key Cross-Reference function block takes the following actions:

1. Verifies that the application-specific participant contains a single key, *not* a composite key.

Determines the participant type of the application-specific participant, which is the application-specific business object. In this business object, verifies that only one key attribute exists. If more than one key attribute exists, the Foreign Key Cross-Reference function block does not know which application-specific key attribute to populate with the application-specific equivalent of the generic business object's foreign key. Therefore, it throws the `RelationshipRuntimeException` exception.

2. Performs the task outlined in Table 111, based on the verb of the application-specific business object.

The Foreign Key Cross-Reference function block obtains from the foreign relationship table an application-specific business object's key value for a given a relationship instance ID *only* if the verb is Update, Delete, or Retrieve. The Foreign Key Cross-Reference function block does *not* obtain the application-specific key value for a Create verb.

Table 111 shows the action that the Foreign Key Cross-Reference function block takes on the foreign relationship table, based on the verb of the generic business object.

Table 111. Actions for the `SERVICE_CALL_REQUEST` calling context and a Foreign Key

Verb of generic business object	Action performed by the Foreign Key Cross-Reference function block
Create	Take no action.  The method writes a new relationship instance to the foreign relationship table when the calling context is <code>SERVICE_CALL_RESPONSE</code> . For more information, see "EVENT_DELIVERY, ACCESS_REQUEST, and SERVICE_CALL_RESPONSE calling contexts" on page 291.
Update Delete Retrieve	<ol style="list-style-type: none"> <li>1. Obtain the generic business object's key value (the relationship instance ID) from the original-request business object in the map execution context.</li> <li>2. Retrieve the relationship instance from the foreign relationship table for the given generic business object's key value. If a relationship instance for this key value does <i>not</i> exist, throw a <code>RelationshipRuntimeException</code> exception. If no participants are found when the verb is Retrieve, throw a <code>CxMissingIDException</code> exception.</li> <li>3. Obtain the application-specific key value from the retrieved relationship instance.</li> <li>4. Copy the application-specific key value into the application-specific business object.</li> </ol>

As Table 111 shows, when the verb is Create, the Foreign Key Cross-Reference function block does *not* write a new relationship instance to the relationship table. It does not perform this write operation because it does not yet have the application-specific foreign key value that corresponds to the instance ID. When the connector processes the application-specific business object, it notifies the application of the need to insert a new row (or rows). If this insert is successful,



the application notifies the connector, which creates the appropriate application-specific business object with a Create verb and the application's key value.

**Note:** For the SERVICE\_CALL\_REQUEST calling context, the Foreign Key Cross-Reference function block manages the foreign relationship table in the same way that the Maintain Simple Identity Relationship function block manages a relationship table.

### **ACCESS\_RESPONSE calling context and foreign keys**

When the calling context is ACCESS\_RESPONSE, the map that is being called is an outbound map; that is, it transforms a generic business object to an application-specific business object. The outbound map receives a generic business object as input and returns an application-specific business object as output. Therefore, the task for the Foreign Key Cross-Reference function block is to obtain from the foreign relationship table the application-specific key for a given generic key value.

For the ACCESS\_RESPONSE calling context, the Foreign Key Cross-Reference function block takes the following actions:

1. Verifies that the application-specific participant contains a single key, *not* a composite key.  
Determines the participant type of the application-specific participant, which is the application-specific business object. In this business object, verifies that only one key attribute exists. If more than one key attribute exists, the Foreign Key Cross-Reference function block does not know which application-specific key attribute to populate with the application-specific equivalent of the generic business object's foreign key. Therefore, it throws the `RelationshipRuntimeException` exception.
2. Locates the relationship instance in the foreign relationship table that matches the value of the foreign key in the generic business object.
3. Obtains the application-specific key value from the retrieved relationship instance.
4. Copies the application-specific key value into the foreign key of the application-specific business object.

The Foreign Key Cross-Reference function block takes these actions on the foreign relationship table regardless of the verb in the generic business object.

## **Tips for using the Foreign Key Cross-Reference and Foreign Key Lookup function blocks**

Keep the following tips in mind when using the Foreign Key Cross-Reference and Foreign Key Lookup function blocks:

- Put the call to the Foreign Key Lookup or Foreign Key Cross-Reference function blocks in the transformation step for the foreign key attribute of the destination business object.
- The Foreign Key Lookup and Foreign Key Cross-Reference function blocks do *not* support composite keys as the foreign key.
- After using the Foreign Key Lookup function block, check that the destination foreign key attribute does *not* contain a null value. A null foreign key value indicates that the Foreign Key Lookup function block was not able to locate the corresponding foreign key value for the foreign key in the source business object. To indicate this condition, log message number 5007 or 5008 (depending



on whether or not the map is forced to fail) and, optionally, throw the `MapFailureException` exception to stop the map.

You do *not* need this check after using the Foreign Key Cross-Reference function block because this function block automatically adds an entry to the foreign relationship table if the application-specific key value does not exist.

- If any of the child object attributes require the use of the Foreign Key Cross-Reference function block or the Foreign Key Lookup function block (but not the Maintain Simple Identity Relationship function block or the Maintain Composite Relationship function block), you can set the verb of the source child object by defining a Move transformation from the source parent object's verb to the child business object's verb. Make the call *inside* the for loop, just before the `runMap()` method is called.

---

## Maintaining custom relationships

The mapping API provides methods to handle operations on relationship tables for some basic relationships such as lookup and identity relationships. For other relationships, you can create custom relationships by programming the adding and deleting of participants yourself.

How you program the transformation code for a relationship attribute depends on several factors that vary with each execution of the map. You typically structure the code as a series of cases handling each of the possible situations that might occur. The following factors are those you most typically need to consider:

- The verb associated with the source business object: Create, Retrieve, Update, or Delete. For example, if the verb is Create, the map usually creates a new relationship instance or adds a new participant instance to an existing relationship instance. Your code should handle each verb that is supported by the application-specific business object and its connector.
- The calling context associated with the map instance. Calling contexts indicate the purpose of the current map execution and can affect how you handle each of the verb cases. For example, if the verb is Create and the calling context is `EVENT_DELIVERY`, you usually create a new relationship instance; if the calling context is `SERVICE_CALL_RESPONSE`, you usually add a participant to the relationship instance. To learn more about calling contexts, see “Understanding map execution contexts” on page 189.
- The business logic contained in the collaboration associated with the map. For example, if the collaboration manages the synchronization of data between two applications, the map developer must coordinate with the collaboration developer to determine which business logic is contained in the collaboration and which is handled in the map.

The `Relationship` and `IdentityRelationship` classes contain methods for creating relationship instances and adding, deleting, and updating participant instances. The `Participant` class provides “set” and “get” methods for specifying and retrieving various properties of participant instances.

The following sections provide information on how to manage a custom relationship:

- “Creating a new relationship instance” on page 296
- “Creating participant instances” on page 296
- “Deleting participant instances” on page 296

## Creating a new relationship instance

To create a new relationship instance, use the `create()` or `addMyChildren()` methods. Both methods create a relationship instance with one new participant instance.

The most common example of creating a new relationship instance is when the verb is `Create` and the calling context is `EVENT_DELIVERY` (or `ACCESS_REQUEST`). In this case, an application generates a “create” event for which a collaboration has subscribed, and the map transforms an application-specific business object to a generic business object. To create a new relationship instance, you must supply the relationship definition name, the participant definition name for the participant you are adding, and the data associated with the participant.

The following code creates a new instance of a relationship called `CustIden` after a customer is added in the Clarify application. The participant definition representing Clarify is called `ClarCust`:

```
instanceId = Relationship.create  
("CustIden", "ClarCust", appBusObj);
```

The return value, `instanceId`, is the instance ID of the new relationship instance.

## Creating participant instances

To create a new participant instance for a relationship, use the `addParticipant()` method. You typically create a new participant instance when:

- The source business object’s verb is `Create` and the calling context is `EVENT_DELIVERY`. In this case, you create a new relationship instance and a new participant instance at the same time.
- The source business object’s verb is `Create` and the calling context is `SERVICE_CALL_RESPONSE`. In this case, you add the new participant to an existing relationship instance.

The following code adds a new participant instance to a relationship called `CustIden`. It assumes a verb of `Create` and a calling context of `SERVICE_CALL_RESPONSE`. The `relID` variable contains the ID of the relationship instance to receive the new participant instance.

```
instanceId = Relationship.addParticipant("CustIden", "SAPCust",  
relID, appBusObj);
```

## Deleting participant instances

To delete a participant instance from a relationship instance, use one of the methods of the `Relationship` class listed in Table 112.

Table 112. Methods that Delete a participant instance

Relationship method	Description
<code>deleteParticipant()</code>	Deletes <i>all</i> participant instances that match the relationship definition name, participant definition name, and participant data that you specify.
<code>deleteParticipantByInstance()</code>	Deletes <i>one</i> participant from a specific relationship instance that you specify.
<code>deactivateParticipant()</code>	Identical to <code>deleteParticipant()</code> except that it leaves a record of the participant in the relationship tables.

Table 112. Methods that Delete a participant instance (continued)

Relationship method	Description
deactivateParticipantByInstance()	Identical to deleteParticipantByInstance() except that it leaves a record of the participant in the relationship tables.

You typically delete participant instances when the source business object has a verb of Delete and the calling context is either EVENT\_DELIVERY (or ACCESS\_REQUEST) or SERVICE\_CALL\_RESPONSE.

**Example:** The following code assumes a calling context of SERVICE\_CALL\_RESPONSE and a verb of Delete. It deletes a participant instance representing a Clarify customer from the CustIden relationship.

```
Relationship.deleteParticipant
("CustIden","ClarCust",appBusObj);
```

## Writing safe relationship code

**Recommendation:** You should use the following defensive coding standards for attributes that require relationship management:

- Always make sure that the source attribute is not null before calling a method of the Mapping API.
- Always put the call to a method of the Mapping API inside the try/catch block and display the appropriate error message inside the catch section.

## Checking for null source attribute

Before calling one of the Mapping API methods in Table 113, make sure that the source attribute is *not* null. If the attribute is null, log an error and do *not* call the method.

Table 113. Handling null source attributes

Mapping API method	Error number to log	Stop map execution?
maintainSimpleIdentityRelationship()	5000	Yes
foreignKeyXref()	5003	The mapping specification should specify whether the map execution should stop.
foreignKeyLookup()		

To stop map execution, you can throw the MapFailureException exception.

## Handling exceptions from the mapping API method

To ensure that any exceptions raised by the Mapping API methods in Table 114 are caught, put the call to a method of the Mapping API inside the try/catch block and log the appropriate error message inside the catch section.

Table 114. Handling Exceptions from mapping API methods

Mapping API method	Exception to Catch	Error Number to Log
maintainSimpleIdentityRelationship()	RelationshipRuntimeException	5001
maintainCompositeRelationship()	CxMissingIDException	5002
foreignKeyLookup()	RelationshipRuntimeException	5007 or 5008
foreignKeyXref()	RelationshipRuntimeException	5009

**Example:** The following code fragment includes a call to the `maintainSimpleIdentityRelationship()` method that catches both the `RelationshipRuntimeException` and `CxMissingIDException` exceptions, logs informational message to display the error text generated by the server, and stops the map execution by throwing `MapFailureException`:

```
try
{
    // API call
    IdentityRelationship.maintainSimpleIdentityRelationship(...);
}

catch (RelationshipRuntimeException re)
{
    logError(5001);
    logInfo(re.toString());
    throw new MapFailureException("RelationshipRuntimeException");
}

catch (CxMissingIDException ce)
{
    logError(5002);
    logInfo(ce.toString());
    throw new MapFailureException("RelationshipRuntimeException");
}
```

**Example:** The following code fragment shows exception handling for the `foreignKeyLookup()` method that catches the `RelationshipRuntimeException` exception, logs informational message to display the error text generated by the server, and then checks the destination attribute to see whether it was successfully mapped; if not, the fragment displays an error with 5007 if the map has to stop execution or message 5008 if it can continue the map execution:

```
try
{
    // API call
    IdentityRelationship.foreignKeyLookup(...);
}

catch (RelationshipRuntimeException re)
{
    logInfo(re.toString());
}

if (ObjDest.isNull("DestAttr")
{
    logError(5007, "DestAttrName", "SrcAttrName", "RelationshipName",
        "ParticipantName", strInitiator);
    throw new MapFailureException("foreignKeyLookup() failed");
}
```

```

If the map execution is to be continued, use the following if statement:
if (ObjDest.isNull("DestAttr")
{
    logError(5008, "DestAttrName", "SrcAttrName", "RelationshipName",
        "ParticipantName", strInitiator);
}

```

**Example:** The following code fragment shows exception handling for the `foreignKeyXref()` method that catches `RelationshipRuntimeException`, logs an informational message to display the error text generated by the server, then checks the destination attribute to see whether it was successfully mapped; if not, the fragment displays an error with message 5009 and stops the map execution by throwing `MapFailureException`:

```

try
{
    // API call
    IdentityRelationship.foreignKeyXref(...);
}

catch (RelationshipRuntimeException re)
{
    logInfo(re.toString());
}

if (ObjDest.isNull("DestAttr")
{
    logError(5009, "DestAttrName", "SrcAttrName", "RelationshipName",
        "ParticipantName", strInitiator);
    throw new MapFailureException("foreignKeyXref() failed");
}

```

---

## Executing queries in the relationship database

As you use relationships, you may need to obtain information about a relationship definition. The relationship information is stored in special tables in the relationship database. To obtain information about a relationship, you can query its relationship 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.

To execute queries in the relationship database:

1. Open a connection to the relationship database to obtain a `DtpConnection` object.
2. Through the `DtpConnection` object, execute queries and manage transactions in the relationship database.

The connection automatically closes when the map finishes execution.

**Important:** Using the `DtpConnection` class and its methods to establish a connection to a relationship database is supported for backward compatibility *only*. These *deprecated methods* will not generate errors, but you should avoid using them and migrate existing code to the new methods. The deprecated methods might be removed in a future release. In new map development, use the `CwDBStoredProcedureParam` class and its methods to obtain a database connection and execute SQL queries. For more information, see “Executing database queries” on page 203.

## Opening a connection

To be able to query the relationship database, you must first open a connection to this database with the `getRelConnection()` method of the `BaseDLM` class. To identify the relationship database to open, specify the name of the relationship definition you want to query. The repository keeps track of the location of the relationship tables for each relationship definition. For more information, see “Advanced settings for relationship definitions” on page 249.

**Example:** The following call to `getRelConnection()` opens the relationship database that contains the relationship tables for the `SapCust` relationship:

```
DtpConnection connection = getRelConnection("SapCust");
```

This call returns a `DtpConnection` object in the `connection` variable, which you can then use to access the relationship database.

## Executing the query

The `executeSQL()` method sends the actual query to the relationship database for execution. This section covers execution of the following kinds of SQL queries:

- Queries that return data from the relationship tables (SELECT)
- Queries that modify the relationship tables (INSERT, UPDATE, DELETE)
- Queries that execute stored procedures

### Queries that return data (SELECT)

The SQL statement `SELECT` queries one or more tables for data. To send a `SELECT` statement to the relationship database for execution, specify a string representation of the `SELECT` as an argument to the `executeSQL()` method.

**Note:** The `DtpConnection.executeSQL()` methods is supported for backward compatibility only. Do *not* use this method in the development of new code; instead, use the `executeSQL()` method of the `CwDBCConnection` class.

**Example:** The following call to `executeSQL()` sends a `SELECT` of one column value from the `ReIRT_T` table:

```
connection.executeSQL(
    "select data from ReIRT_T where INSTANCEID = 2");
```

**Note:** In the preceding code, the `connection` variable is a `DtpConnection` object obtained from a previous call to the `getRelConnection()` method.

You can also send a `SELECT` statement that has parameters in it by using the second form of the `executeSQL()` method.

**Example:** The following call to `executeSQL()` performs the same task as the previous example except that it passes the instance ID as a parameter to the `SELECT` statement:

```
Vector argValues = new Vector();

String instance_id = "2";
argValues.addElement( instance_id );
connection.executeSQL(
    "select data from ReIRT_T where INSTANCEID = ?", argValues);
```

The `SELECT` statement returns data from the relationship tables as rows. Each row is one row from the specified relationship table that matches the conditions in 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 relationship database you are accessing. Consult your database documentation for the exact syntax of the SELECT statement.

To access the returned data, perform the following steps:

1. Obtain one row of data.
2. Obtain column values, one by one.

Table 115 shows the methods in the `DtpConnection` class that provide access to the rows of returned query data.

*Table 115. DtpConnection methods for row access*

Row-Access task	DtpConnection 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. See Table 71 on page 206 for the Java methods for accessing the columns of a returned query row.

**Note:** The mechanism for accessing rows from the query result is the same for the deprecated `DtpConnection` class as for its replacement, the `CwDBCConnection` class. For more information, see “Executing static queries that return data (SELECT)” on page 205.

**Example:** The following code sample gets an instance of the `DtpConnection` class, which is a connection to the relationship database that stores the `sampleRelationshipName` relationship definition. It then executes a `SELECT` statement that returns only one column with a single string value of “CrossWorlds”:

```
Vector theRow = null;
Enumeration theRowEnum = null;
String theColumn1 = null;
DtpConnection connectn = null;

try
{
    connectn = getRelConnection("sampleRelationshipName");
}

catch(DtpConnectionException e)
{
    System.out.println(e.getMessage());
}

// Test for a resulting single column, single row, result set
// specified condition
try
{
```

```

connectn.executeSQL(
    "select data from Re1RT_T where INSTANCEID = 2");

// Loop through each row
while(connectn.hasMoreRows())
{
    theRow = connectn.nextRow();
    int length = 0;
    if ((length = theRow.size())!= 1)
    {
        return "Expected result set size = 1," +
            " Actual result state size = " + length;
    }

    // Get each column
    theRowEnum = theRow.elements();
    if(theRowEnum.hasMoreElements())
    {
        // Get the value
        theColumn1 = (String)theRowEnum.nextElement();
        if(theColumn1.equals("CrossWorlds")==false)
        {
            return "Expected result = CrossWorlds,"
                + " Resulting result = " + theColumn1;
        }
    }
}

}

catch(DtpConnectionException e)
{
    System.out.println(e.getMessage());
}
}

```

**Note:** The SELECT statement does *not* modify the contents of the relationship database. Therefore, you do *not* usually need to perform transaction management for SELECT statements.

### Queries that modify the relationship tables

SQL statements that modify a relationship table include the following:

- INSERT adds new rows to a relationship table.
- UPDATE modifies existing rows of a relationship table.
- DELETE removes rows from a relationship table.

To send one of these statements to the relationship database for execution, specify a string representation of the statement as an argument to the executeSQL() method.

**Note:** The DtpConnection.executeSQL() methods is supported for backward compatibility only. Do *not* use this method in the development of new code; instead, use the executeSQL() method of the CwDBCConnection class.

**Example:** The following call to executeSQL() sends an INSERT of one row into the Re1RT\_T table:

```

connection.executeSQL("insert into Re1RT_T values
(1, 3, 6)");

```

**Note:** In the preceding code, the connection variable is a DtpConnection object obtained from a previous call to the getRe1Connection() method.



For an UPDATE or INSERT statement, you can determine the number of rows in the relationship table that have been modified or added with the `getUpdateCount()` method.

Because the INSERT, UPDATE, and DELETE statements modify the contents of the relationship database, you *must* perform transaction management for these statements. 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. Table 116 shows the methods in the `DtpConnection` class that provide transaction support for SQL queries.

Table 116. *DtpConnection* methods for transaction management

Transaction-Management task	DtpConnection method
Begin a new transaction.	<code>beginTran()</code>
End the transaction, committing (saving) all changes made during the transaction to the database.	<code>commit()</code>
Determine if a transaction is currently active.	<code>inTransaction()</code>
End the transaction, rolling back (backing out) all changes made during the transaction.	<code>rollback()</code>

To mark the beginning of a transaction in the relationship database, use the `beginTran()` method. Execute all SQL statements that must succeed or fail as a unit *between* this call to `beginTran()` and the end of the transaction. You can 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 relationship database.
- Call `rollback()` to end the transaction unsuccessfully. All modifications that the SQL statements have made are *backed out* of the relationship 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.

```
DtpConnection connection = getRelConnection("SapCust");
```

```
// begin a transaction
connection.beginTran();

// insert a row
connection.executeSQL("insert...");

// commit the transaction
connection.commit();

// release the database connection
releaseRelConnection(true);
```

To determine whether a transaction is currently active, use the `inTransaction()` method.

### Queries that call stored procedures

A stored procedure is a user-defined procedure that contains SQL statements and conditional logic. Stored procedures are stored in a database. When you create a new relationship, Relationship Designer creates a stored procedure to maintain each relationship table.

Table 117 shows the methods in the `DtpConnection` class that call a stored procedure. These methods are supported for backward compatibility only. Do *not* use these methods in the development of new code; instead, use the `executeSQL()` and `executeStoredProcedure()` methods of the `CwDBConnection` class.

Table 117. *DtpConnection* methods for Calling a Stored Procedure

How to call the stored procedure	<code>DtpConnection</code> method	Use
Send a CALL statement that executes the stored procedure to the relationship database.	<code>executeSQL()</code>	To call a stored procedure that does <i>not</i> have OUT parameters
Specify the name of the stored procedure and an array of its parameters to create a procedure call, which is sent to the relationship database for execution.	<code>execStoredProcedure()</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 Mapping API provides is simpler and it reuses database resources, which can increase the efficiency of execution. You should use the Mapping API to execute stored procedures.

As Table 117 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 `execStoredProcedure()` to call the stored procedure.
- The number of times you call the stored procedure  
The `execStoredProcedure()` method precompiles the stored procedure. Therefore, if you call the same stored procedure more than once (for example, in a loop), use of `execStoredProcedure()` can be faster than `executeSQL()` because the relationship database can reuse the precompiled version.

The following sections describe how to use the `executeSQL()` and `execStoredProcedure()` methods to call a stored procedure.

**Calling stored procedures with `executeSQL()`:** To call a stored procedure with the `executeSQL()` method, specify as an argument to the `executeSQL()` method a string representation of the CALL statement that includes the stored procedure and any arguments.

**Example:** The following call to `executeSQL()` sends a CALL statement to execute the `setOrderCurrDate()` stored procedure:

```
connection.executeSQL("call setOrderCurrDate(345698)");
```

**Note:** In the preceding code, the `connection` variable is a `DtpConnection` object obtained from a previous call to the `getRelConnection()` method.

You can execute the `setOrderCurrDate()` stored procedure because its single argument is an IN parameter; that is, the value is only sent into the stored procedure. The stored procedure does not have any OUT parameters.

**Note:** You can use the form of `executeSQL()` that accepts a parameter array to pass in parameter values. However, you *cannot* use `executeSQL()` to call a stored procedure that uses an OUT parameter. To execute such a stored procedure, you *must* use `execStoredProcedure()`.

Use an anonymous PL/SQL block if you plan on calling Oracle stored PL/SQL objects via ODBC using the `DtpConnection.executeSQL()` method. The following is an acceptable format (the stored procedure name is `myproc`):

```
executeSQL("begin myproc(...); end;");
```

**Calling stored procedures with `execStoredProcedure()`:** To call a stored procedure with the `execStoredProcedure()` method, you:

1. Specify the name of the stored procedure to execute as a string.
2. Build a `Vector` parameter array of `UserStoredProcedureParam` objects, which provide parameter information (such as the name, type, and value of each 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 IN/OUT 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 `UserStoredProcedureParam` object describes a single parameter for a stored procedure. Table 118 shows the parameter information that a `UserStoredProcedureParam` object contains as well as the methods to retrieve and set this parameter information.

*Table 118. Parameter information in a `UserStoredProcedureParam` Object*

Parameter information	<code>UserStoredProcedureParam</code> method
Parameter name	<code>getParamName()</code> , <code>setParamName()</code>
Parameter value	<code>getParamValue()</code> , <code>setParamValue()</code>
Parameter data type:	
• As a Java Object	<code>getParamDataTypeJavaObj()</code> , <code>setParamDataTypeJavaObj()</code>
• As a JDBC data type	<code>getParamDataTypeJDBC()</code> , <code>setParamDataTypeJDBC()</code>
Parameter in/out type	<code>getParamIOType()</code> , <code>setParamIOType()</code>
Parameter index position	<code>getParamIndex()</code> , <code>setParamIndex()</code>

**Steps for passing parameters to a stored procedure:** To pass parameters to a stored procedure, perform the following steps:

1. Create a `UserStoredProcedureParam` object to hold the parameter information. Use the `UserStoredProcedureParam()` constructor to create a new `UserStoredProcedureParam` object. To this constructor, pass the following parameter information to initialize the object:

- Parameter index indicates the position within the declaration of the stored procedure for this parameter.
  - Parameter data type is usually the name of the Java Object that holds the parameter value.
  - Parameter value is a Java Object that contains the value to assign to the parameter. For an OUT parameter, this value can be a dummy value but the Object type should correspond to the OUT parameter data type in the stored-procedure declaration.
  - Parameter in/out type specifies whether the parameter is an IN, INOUT, or OUT parameter.
  - Parameter name associates a string name with the parameter.
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 UserStoredProcedureParam object to the parameter Vector object.  
Use the addElement() method of the Vector class to add the UserStoredProcedureParam object.
  5. Once you have created all UserStoredProcedureParam objects and added them to the Vector parameter array, pass this parameter array as the second argument to the execStoredProcedure() method.  
The execStoredProcedure() method sends the stored procedure and its parameters to the relationship database for execution.

**Note:** The first argument to execStoredProcedure() is the name of the stored procedure to execute.

**Example:** Suppose you have the get\_empno() stored procedure defined 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 stored procedure has two parameters:

- The first parameter, emp\_id, is an IN parameter.  
Therefore, you must initialize its associated UserStoredProcedureParam object with an in/out type of "IN", as well as with the appropriate data type, name, and the 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 data type and value must be of a Java Object that holds integer values: Integer.
- The second parameter, emp\_number, is an OUT parameter.  
For this parameter, create an *empty* UserStoredProcedureParam object to send into the stored procedure. You initialize this object with an in/out type of "OUT" as well as with the appropriate data type and name. However, you provide a dummy value for this parameter. Once the stored procedure completes execution, you can obtain the returned value from this OUT parameter with the getParamValue() method.

**Example:** The following example executes the get\_empno() stored procedure with the execStoredProcedure() method:

```

DtpConnection connectn = null;
try
{
    // Get database connection
    connectn = getRelConnection("Customer");

    // Create parameter Vector
    Vector paramData = new Vector(2);

    // Construct the procedure name
    String sProcName = "get_empno";

    // Create IN parameter
    UserStoredProcedureParam arg_in = new UserStoredProcedureParam(
        1, "Integer", new Integer(6), "IN", "arg_in");

    // Create dummy argument for OUT parameter
    UserStoredProcedureParam arg_out = new UserStoredProcedureParam(
        2, "Integer", new Integer(0), "OUT", "arg_out");

    // Add these two parameters to the parameter Vector
    paramData.addElement(arg_in);
    paramData.addElement(arg_out);

    // Run get_empno() stored procedure
    connectn.execStoredProcedure(sProcName, paramData);

    // Get the result from the OUT parameter
    arg_out = (UserStoredProcedureParam) paramData.elementAt(1);
    Integer emp_number = (Integer) arg_out.getParamValue();
}

```

**Tip:** The Vector object is a zero-based array while the UserStoredProcedureParam objects are indexed as a one-based array. In the preceding code, the OUT parameter is created with an index value of 2 in the UserStoredProcedureParam() constructor because this parameter array is one-based. However, to access the value for this OUT parameter from the Vector parameter array, the elementAt() 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 Mapping API must convert a parameter value between these two data types. For an IN parameter, the Mapping API converts the parameter value from a Java Object to its SQL data type. For an OUT parameter, the Mapping API converts the parameter value from its SQL data type to a Java Object. The Mapping API converts a parameter value between these two data types using two layers of data type mapping:

- From Java type to JDBC type
- From JDBC type to SQL data type

The Mapping API 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.

To map an IN (or INOUT) parameter from a Java object to its JDBC equivalent, the Mapping API uses the mappings in Table 119.

Table 119. Mappings from Java object to JDBC data type equivalent

From Java object	To JDBC data type
String	CHAR, VARCHAR, or LONGVARCHAR
java.math.BigDecimal	NUMERIC
Boolean	BIT
Integer	INTEGER
Long	BIGINT
Float	REAL
Double	DOUBLE
byte[]	BINARY, VARBINARY, or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
Clob	CLOB
Blob	BLOB
Array	ARRAY
Struct	STRUCT
Ref	REF

To map an OUT (or INOUT) parameter from a JDBC data type to its Java object equivalent, the Mapping API uses the mappings in Table 120.

Table 120. Mappings from JDBC data type to Java object

From JDBC data type	To Java object
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	Integer
SMALLINT	Integer
INTEGER	Integer
BIGINT	Long
REAL	Float
FLOAT, DOUBLE	Double
BINARY, VARBINARY, or LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	Clob
BLOB	Blob
ARRAY	Array
STRUCT	Struct
REF	Ref

Therefore, every `UserStoredProcedureParam` object contains two representations of its data type, as Table 121 shows.

Table 121. Parameter data types

Parameter data type	Description	UserStoredProcedureParam method
The Java Object	Data type that the map-transformation code uses to hold the parameter value	<code>getParamDataTypeJavaObj()</code> , <code>setParamDataTypeJavaObj()</code>
The JDBC data type	Data type that the Mapping API uses internally to hold the parameter value	<code>getParamDataTypeJDBC()</code> , <code>setParamDataTypeJDBC()</code>

You can use the `UserStoredProcedureParam` methods in Table 121 to access either form of the parameter data type. However, you should use the Java Object data type (such as `Integer`, `String`, or `Float`) for the following reasons:

- For IN (and INOUT) parameters, you *must* provide the parameter value as a Java Object. Therefore, providing the parameter data type as a Java Object is more consistent.
- The `execStoredProcedure()` method sends parameters in a `Vector` parameter array. The `Vector` object can only contain elements that are Java Objects.

The `emp_id` parameter of `get_empno()` is declared with the SQL data type of `NUMBER`, which contains an integer value. Therefore, in the code example that starts on 307, the call to `UserStoredProcedureParam()` for the `emp_id` parameter (parameter with the index position of 1) sets its value to 6 with its third argument of:

```
new Integer(6)
```

This call also sets the parameter type to the same Java Object type with its second argument of:

```
"Integer"
```

## Closing a connection

The connection to the relationship database is released when the map is finished executing. If the map executes successfully, all transactions are automatically committed if they are not already explicitly committed. If the map execution fails (for instance, if an exception is thrown that is not handled with a catch statement), all transactions are rolled back if they are not already explicitly rolled back.

---

## Loading and unloading relationships

With the `repos_copy` utility, you can load and unload specified relationship definitions in the repository.

**Note:** You can also use `repos_copy` to load and unload map definitions in the repository. For more information, see “Importing and exporting maps from InterChange Server” on page 79.

## Unloading a relationship definition

With the `repos_copy` utility, you can unload specified relationship definitions in the repository with the `-e` option. A *relationship repository file* is the file that the `repos_copy` utility creates when it extracts a relationship definition from the repository into a text (`.jar`) file.

**Example:** The following `repos_copy` command unloads the `StateLk` relationship definition from the repository of an InterChange Server named `WebSphereICS` into a relationship repository file:

```
repos_copy -eRelationship:StateLk -oRL_StateLookup.jar  
-sWebSphereICS -uadmin -pnull
```

**Attention:** A relationship is *not* a first-class entity. Therefore, its name space is separate from the first-class entities. While no first-class entities can have the same name, a relationship can have the same name as a first-class entity (such as a business object or collaboration). However, if a relationship definition has a name that matches *any* existing first-class entity, you cannot use the `-e` option of `repos_copy` to unload or load



that relationship definition. You can load and unload the entire repository, which includes relationship definitions.

You can copy several relationship definitions into one relationship repository file.

**Example:** To copy both the StateLk and CustLkUp relationship definitions, use the following `repos_copy` command:

```
repos_copy -eRelationship:StateLk+Relationship:CustLkUp
-oRL_Lookup_Relationships.jar -sWebSphereICS -uadmin -
pnull
```

## Loading a relationship definition

You can also use `repos_copy` to load a relationship definition into the repository from a relationship repository file.

**Example:** The following `repos_copy` command loads the StateLk relationship definition into the repository of an InterChange Server named WebSphereICS:

```
repos_copy -iRL_StateLookup.jar
-sWebSphereICS -uadmin -pnull
```

The `repos_copy` utility performs the following validations when it loads a relationship definition:

- It validates the Database URL of the relationship definition it loads.
- It validates that any dependent objects for the relationship definition already exist in the repository.

If `repos_copy` cannot perform both of these validations, it cannot load the relationship definition. However, `repos_copy` provides special command-line options to suppress or restrict these validations, as the following sections explain.

### Validating the database URL

The `repos_copy` utility provides the `-r` option to assist in loading relationship definitions into a repository. The `-r` option tells `repos_copy` to add relationship definitions to the repository without creating their run-time schemas. When `repos_copy` backs up an entire repository (with the `-o` option), some of the information in the resulting repository text file describes relationship definitions. If you then use `repos_copy` (without the `-r` option) to load a different repository with the contents of this repository text file, `repos_copy` might generate errors of the following format when it attempts to load the relationship definitions:

```
Server error: An error occurred during the validation
of the runtime database connection information for
relationship definition Customer. The database URL
used is: jdbc:weblogic:mssqlserver4:Cwrelns312@CWDEV:1433.
The database login name used is: crossworlds.
The database type used is: W55s/wPE/14=1.
Reason: SqlServer.
```

The cause of this error is `repos_copy`'s attempt to validate the URL for the relationship database. Part of a relationship's definition is the Database URL of the relationship database.

If `repos_copy` cannot find the relationship database, it generates an error and rolls back the repository load. If you are just backing up and restoring on the same InterChange Server (with the same relationship databases), you do not need to include the `-r` option. Validation of the relationship database URL succeeds



because the database URLs can be located. Therefore, the repository load (including the relationship definitions) is successful.

However, in the import process of a migration when you are moving repository data from one machine to another, the `-r` option can be helpful. If you execute the `repos_copy` command in an environment that cannot locate any existing relationship databases in the repository data, `repos_copy` generates the validation error. To suppress this validation, include the `-r` option of `repos_copy` when you load the repository. By suppressing this validation, `repos_copy` can successfully add the relationship definitions to the repository. It uses the current repository database as the location for the relationship database. You can then use Relationship Designer to change the Database URL to point to the appropriate location of each relationship database.

**Example:** The following `repos_copy` command loads the StateLk relationship definition into the repository, suppressing the validation of its Database URL:

```
repos_copy -rStateLk -iRL_StateLookup.txt -sWebSphereICS -uadmin  
-pnul1
```

### Validating dependent objects

By default, `repos_copy` validates whether all dependent objects exist when it loads a relationship definition. For example, it checks that all business objects involved in the relationship exist in the repository. If all dependent objects do *not* exist, `repos_copy` generates an error and rolls back the repository load. In the `repos_copy` command window, the following message is displayed:

```
Some of the participants for relationships were missing.  
For more info, refer to InterChange Server log file.
```



---

## Part 3. Mapping API Reference



---

## Chapter 9. BaseDLM class

The methods documented in this chapter operate on map instances. They are defined on the IBM WebSphere InterChange Server-defined class BaseDLM. The BaseDLM class is the base class for all map instances. All created maps are subclasses of BaseDLM; they all inherit these methods. The BaseDLM class provides utility methods for error handling and debugging in maps, and establishing a connection to a database. All methods in this class can be called without referring to the class name.

Table 122 summarizes the methods of the BaseDLM class.

Table 122. BaseDLM method summary

Method	Description	Page
getDBConnection()	Establishes a connection to a database and returns a CwDBConnection object.	315
getName()	Retrieves the name of the current map.	317
getRelConnection()	Establishes a connection to a relationship database and returns a DtpConnection object.	318
implicitDBTransactionBracketing()	Retrieves the transaction programming model that the map instance uses for any connection it obtains.	319
isTraceEnabled()	Compares the specified trace level with the current trace level of the map.	319
logError(), logInfo(), logWarning()	Sends an error, information, or warning message to the InterChange Server log file.	320
raiseException()	Raises an exception.	321
releaseRelConnection()	Releases a connection to a relationship database.	323
trace()	Generates a trace message.	324

---

### getDBConnection()

Establishes 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 `CwDBConnection` object.

## Exceptions

`CwDBConnectionFactoryException` – If an error occurs while trying to establish the database connection.

## Notes

The `getDBConnection()` 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 that connection. All connections in a particular connection pool are associated with the same database. The method returns a `CwDBConnection` object through which you can execute queries and manage transactions on the database. See the methods in the `CwDBConnection` 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 `getDBConnection()` method. The following `getDBConnection()` call specifies explicit transaction bracketing for the connection obtained from the `ConnPool` connection pool:

```
conn = getDBConnection("ConnPool",false);
```

The connection is released when the map instance 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.

## 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 in a table of the database.

```
CwDBConnection connection = getDBConnection("CustConnPool");
```

```
// Insert a row  
connection.executeSQL("insert...");
```

```
// Update rows...  
connection.executeSQL("update...");
```

Because the preceding call to `getDBConnection()` 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 Map Properties dialog). 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 13, "CwDBConnection class", `implicitDBTransactionBracketing()`, `isActive()`, `release()`

---

## getName()

Retrieves the name of the current map.

### Syntax

```
String getName()
```

### Parameters

None.

### Return values

None.

### Exceptions

None.

### Examples

The following example obtains the name of the current map and logs an informational message:

```
String mapName = getName();
logInfo(mapName + " is starting");
```

---

## getRelConnection()

Establishes a connection to a relationship database and returns a `DtpConnection` object.

### Syntax

```
DtpConnection getRelConnection(String relDefName)
```

### Parameters

*relDefName*      A relationship definition name. The method connects to the database containing the relationship tables for this relationship definition.

### Return values

Returns a `DtpConnection` object.

### Exceptions

`DtpConnectionException` – If an error occurs while trying to establish the database connection.

### Notes

This method establishes a connection to the database that contains the relationship tables used by the *relDefName* relationship, and provides a way to perform queries and updates to the relationship database. The method returns a `DtpConnection` object through which you can execute queries and manage transactions. See the methods in the `DtpConnection` class for more information.

The connection is released when the map is finished executing. You can explicitly close this connection with the `releaseRelConnection()` method.

### Examples

The following example establishes a connection to the database containing the relationship tables for the `SapCust` relationship. It then uses a transaction to execute a query for inserting rows into a table in the `SapCust` relationship.

```
DtpConnection connection = getRelConnection("SapCust");

// begin a transaction
connection.beginTran();

// insert a row
connection.executeSQL("insert...");

// update rows...
connection.executeSQL("update...");

// commit the transaction
connection.commit();
```

### See also

`getDBConnection()`, Chapter 15, "DtpConnection class", `releaseRelConnection()`



---

## implicitDBTransactionBracketing()

Retrieves the transaction programming model that the map instance 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 map instance 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 map instance uses explicit transaction bracketing for the database associated with the `conn` connection:

```
if (implicitDBTransactionBracketing())  
    CwDBConnection conn = getDBConnection("ConnPool", false);
```

### See also

```
getDBConnection()
```

---

## isTraceEnabled()

Compares the specified trace level with the current trace level of the map.

### Syntax

```
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 whether or not to log a trace message. 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()

Sends an error, information, or warning message to the InterChange Server log file.

### 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

<i>message</i>	The message text.
<i>messageNum</i>	The number of a message in a message text file.
<i>param</i>	A single parameter. There can be up to five parameters, separated by commas. Each is sequentially resolved to a parameter in the message text.
<i>paramArray</i>	An array of parameters.

### Return values

None.

### Exceptions

None.

### Notes

This method sends a message to the InterChange Server's logging destination. The logging destination can be a file, a window, or both.

By default, the logging destination is the file `InterchangeSystem.log`. You can change the logging destination by entering a value for the `LOG_FILE` parameter in the configuration file, `InterchangeSystem.cfg`. The parameter value can be a file name, `STDOUT` (which writes the log to the server's command window), or both.

Within each set of methods:

- The first form is self-contained and includes all of the text necessary to generate a message.
- The second form generates a message that does not have parameters.
- The third form contains a message number and a set of parameter values.
- The fourth form uses an array of parameters.

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 Appendix A, "Message files," on page 497.

## Examples

The following example logs an informational message, using `getString()` to obtain an attribute value to log in the message.

```
logInfo("Item shipped. CustomerID: "  
+ fromCustomerBusObj.getString("CustomerID"));
```

The following example logs an error message whose text is contained in the map 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);
```

## See also

`trace()`

---

## raiseException()

Raises an exception.

### Syntax

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

```
void raiseException(String exceptionType, int messageNum,  
String parameter[,...])
```

```
void raiseException(RuntimeEntityException exception)
```

### Parameters

*exceptionType* One of the following IBM WebSphere InterChange Server-defined constants:

	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 that is not part of the IBM WebSphere InterChange Server API.
	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 IBM WebSphere InterChange Server system.
<i>message</i>		A text string that embeds the exception message in the method call.
<i>messageNum</i>		A reference to a numbered message in the map message file.
<i>parameters</i>		A value for the parameter in the message itself. There can be up to five parameters in the method call.
<i>exception</i>		The name of an exception object variable.

## Return values

None.

## Notes

The `raiseException()` method has three forms:

- The first form of the method creates a new exception, passing an exception type and a string. Use it to embed a message into the method call itself.
- The second form creates a new exception, passing an exception type and a reference to a message in the map message file. The method call can contain up to five parameters, separated with commas.
- The third form raises an exception object that the map has previously handled. For example, a transformation step might get an exception, assign it to a variable, and do some other work. Finally, the transformation step raises the exception.

**Note:** 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 Appendix A, “Message files,” on page 497.

## Examples

The following example uses the first form of the method to raise an exception of `ServiceCallException` type. The text is embedded in the method call.

```
raiseException(ServiceCallException,  
    "Attempt to validate Customer failed.");
```

The next example raises an exception of `ServiceCallException` type. The message in the message file is as follows:

```
23  
Customer update failed for CustomerID={1} CustomerName={2}
```

The `raiseException()` method invokes the message, retrieves the values of the message parameters from the `fromCustomer` variable, and passes them to the `raiseException()` call.

```
raiseException(ServiceCallException, 23,  
    fromCustomer.getString("CustomerID"),  
    fromCustomer.getString("CustomerName"));
```

The final example raises a previously handled exception. The system-defined variable `currentException` is an exception object that contains the exception.

```
raiseException(currentException);
```

---

## releaseRelConnection()

Releases a connection to a relationship database.

### Syntax

```
void releaseRelConnection(Boolean doCommit)
```

### Parameters

<i>doCommit</i>	The flag that indicates whether this method should call the <code>DtpConnection.commit()</code> method before it releases the database connection.
-----------------	--

### Return values

None.

### Exceptions

`DtpConnectionException` – If an error occurs while trying to release the database connection or if the requested commit or rollback has failed.

### Notes

The `releaseRelConnection()` method releases the connection for this specific map. It commits or rolls back the database transactions based on the value of its `doCommit` argument, as follows:

- If `doCommit` is true, `releaseRelConnection()` assumes it was called after the successful completion of the operation on a database and therefore it is safe to commit the transaction.
- If `doCommit` is false, `releaseRelConnection()` assumes it was called as the result of an exception and therefore the transaction must be rolled back.

Once `releaseRelConnection()` has performed the chosen action on the database transaction, it releases the database connection that the current thread is exclusively using.

## See also

`getRelConnection()`, `release()`

---

## trace()

Generates a trace message.

## 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 causes the message to be generated.
<i>traceMsg</i>	A string that prints to the trace file.
<i>messageNum</i>	A number that represents a message in the map message file.
<i>param</i>	A single parameter. You can add additional single parameters, separated by commas, up to a total of five.
<i>paramArray</i>	An array of parameters.

## Notes

The `trace()` method generates a message that the map prints if tracing is turned on. This method has five forms:

- The first form takes just a string message that appears when tracing is set to any level.
- 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 map 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 map 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 map message file, and an array of parameter values.

**Note:** 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 Appendix A, “Message files,” on page 497.

You can set the trace level for a map as part of the Map Properties.

## Examples

The following example generates a Level 2 trace message and supplies the text of the message:

```
trace (2, "Starting to trace at Level 2");
```

The following example prints message 201 in the map message file if the 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");
```

## See also

```
logError(), logInfo(), logWarning()
```





---

## Chapter 10. BusObj class

The methods documented in this chapter operate on objects of the BusObj class.

**Note:** The BusObj class is used for both collaboration development and mapping; check the Notes section for each method's usage issues.

The first two sections of this chapter explain the exceptions listed with these methods and how to specify attributes and child business objects in a hierarchical business object. The rest of the sections describe the methods listed in Table 123.

*Table 123. BusObj method summary*

Method	Description	Page
copy()	Copy all attribute values from the input business object to this one.	329
duplicate()	Create a business object (BusObj object) exactly like this one.	330
equalKeys()	Compare this business object's key attribute values with those in the input business object.	330
equals()	Compare this business object's attribute values with those in the input business object, including child business objects.	331
equalsShallow()	Compare this business object's attribute values with those in the input business object, excluding child business objects from the comparison.	332
exists()	Check for the existence of a business object attribute with a specified name.	332
getBoolean(), getDouble(), getFloat(), getInt(), getLong(), get(), getBusObj(), getBusObjArray(), getLongText(), getString() and getLocale()	Retrieve the value of a single attribute from a business object.	333
getType()	Retrieve the locale of the business object's data.	335
getVerb()	Retrieve the name of the business object definition on which this business object was based.	335
isBlank()	Retrieve this business object's verb.	336
isKey()	Find out whether the value of an attribute is set to a zero-length string.	336
isNotNull()	Find out whether a business object's attribute is defined as a key attribute.	336
isRequired()	Find out whether the value of a business object's attribute is null.	337
keysToString()	Find out whether a business object's attribute is defined as a required attribute.	338
set()	Retrieve the values of a business object's primary key attributes as a string.	338
	Set a business object's attribute to a specified value of a particular data type.	339

Table 123. BusObj method summary (continued)

Method	Description	Page
setContent()	Set the contents of this business object to another business object.	340
setDefaultAttrValues()	Set all attributes to their default values.	341
setKeys()	Set the values of this business object's key attributes to the values of the key attributes in another business object.	341
setLocale()	Set the locale of the current business object.	342
setVerb()	Set the verb of a business object.	342
setVerbWithCreate()	Create the instance of the child business object and set its verb.	343
setWithCreate()	Set a 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.	343
toString()	Return the values of all attributes in a business object as a string.	344
validData()	Checks whether a specified value is a valid type for a specified attribute.	345

## Exceptions and exception types

Methods for which exceptions or exception types are listed throw the `CollaborationException` exception. Some methods have both exceptions and exception types listed. Both of these relate to a `CollaborationException` object and differ as follows:

- An *Exception* is a class that is subclassed from `CollaborationException`. If there is a subclassed exception, you can use it in mapping to determine more closely the cause of the problem.
- An *Exception type* is a piece of data in a `CollaborationException` object. Collaboration developers use this exception type to catch exceptions through the Designer user interface. In addition, all users of `BusObj` can use this field to determine the reason for a failure if there is no exception class thrown that is more detailed than `CollaborationException`.

## Syntax for traversing hierarchical business objects

When you are writing code that requires that you traverse hierarchical business objects, you need to use the syntax that lets you specify attributes in elements in child business object arrays that are elements of child business object arrays, and other such complexities. This chapter specifies the syntax to use.

An attribute specification can be:

```
[[attributeName[index].]...]attributeName
```

This syntax expands to any of the following formats:

```
attributeName
attributeName[index].attributeName
attributeName[index]... .attributeName
```

**Note:** Do not use the period (.) when creating a business object attribute name. If a business object attribute has a period within its name, an IBM WebSphere

InterChange Server map interprets the period as Java's dot operator and imparts special meaning to it. For example, "attribute.name" will be interpreted as "name" being a field or method for the "attribute" object.

## Specifying an attribute of basic type

The following example uses the `busObj.get()` method to retrieve a basic type attribute named `OrderID` from the business object `orderObj`.

```
orderObj.get("OrderID");
```

## Specifying an attribute in a child business object

The following example assumes that `orderObj` is a hierarchical business object. One of its attributes is `CustomerInfo`, a single-cardinality child business object. The example retrieves the customer name from the `CustomerName` attribute of `CustomerInfo`.

```
orderObj.get("CustomerInfo.CustomerName");
```

## Specifying an attribute in a child of a child business object

If there is a chain of child business objects, in which `CustomerInfo` is a child of `orderObj` and `AddressInfo` is a child of `CustomerInfo`, you can retrieve city information from `AddressInfo` as follows:

```
orderObj.get("CustomerInfo.AddressInfo.City");
```

## Specifying an attribute in an element of an array of child business objects

You can also refer to a child business object in an array by specifying its index in the array. The first element in the array always begins with zero. For example, the following example retrieves the value of the `Quantity` attribute from the third child business object in an array.

```
orderObj.get("LineItem[2].Quantity");
```

---

## copy()

Copy all attribute 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

`equalsShallow()`, `equals()`

## 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.

**Note:** Passing in the business object as an `Object` ensures that this `equals()` method overrides the `Object.equals()` method.

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.

## See also

`equalsShallow()`, `equalKeys()`

## 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);
```

---

## `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

`equals()`, `equalKeys()`

## 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; otherwise, 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

<i>attribute</i>	The name of an attribute.
<i>position</i>	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

The `get()` method retrieves an attribute value from the current business object. It 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 the value that `get()` returns. Each time this method is called, it returns a new copy (clone) of the attribute.

The `get()` method provides 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.

### Returns

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

In collaboration development, 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.

### Returns

Returns true if the attribute value is a zero-length string; returns false otherwise.

### 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 keyAttr = (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.

At instantiation of a new business objects, all attribute values are initialized with a 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 is processing a business object received from a connector, the mapping process might have inserted the null. 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#*, this could be the output:

```
SS#=100408394
```

If the primary key attributes are *FirstName* and *LastName*, this could be the output:

```
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

The `set()` method sets an attribute value in the current business object. This method sets an object reference to the *value* parameter 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);
    ...
}
```

---

## setContent()

Set the contents of this business object to another business object. Where both business objects *share* the content.

## Syntax

```
void setContent(BusObj BusObj)
```

## Notes

Using `setContent()` is similar to using `copy()` in that both functions copy the content from one business object to another. But `setContent()` *shares* the content of both business objects. For example, if the content in `BusObjA` is shared with `BusObjB`, then if the content in `BusObjA` changes, then the content in `BusObjB` also changes at the same time.

## Parameters

*BusObj*            The business object whose values are used to set values of this business object.

## Exceptions

`CollaborationException`—The `setContent()` 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 contents of the instance variable for the output object `ObjOutput1` to the contents of the business object `rDstB0[0]`.

```
ObjOutput1.setContent(rDstB0[0]);
```

---

## 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. The 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

The `setVerb()` method is used only in mapping.



**Note:** Do *not* use this method in collaboration development, 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");`

---

## setVerbWithCreate()

Create the instance of the child business object and set its verb.

### Syntax

```
void setVerbWithCreate(String attributeName, String verb)
```

### Parameters

*attributeName* The name of the child business object created

*verb* The verb to be set.

### Exceptions

CollaborationException—The `setVerbWithCreate()` method can set the following exception type for this exception:

- AttributeException—Set if an attribute access problem occurs.

### Notes

If the attribute specified by the *attributeName* parameter 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* parameter. If the instance of this child business object already exists, only its verb is set. If the child business object is of multi-cardinality, the *attributeName* parameter should specify the subscript.

## Examples

The following example creates an instance of the `childBO` child business object and sets its verb to Create:

```
myBO.setVerbWithCreate("childBO", "Create");
```

---

## setWithCreate()

Set a 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.

### Syntax

```
void setWithCreate(String attributeName, BusObj busObj)  
void setWithCreate(String attributeName, BusObjArray busObjArray)  
void setWithCreate(String attributeName, Object value)
```

### Parameters

*attributeName* The name of the attribute to set.

*busObj* The business object to insert into the target attribute.

*busObjArray* The business object array to insert into the target attribute.  
*value* The object to insert into the target attribute. This object needs to be one of the following types: BusObj, BusObjArray, Object.

## 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 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 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 methods

Some methods in the BusObj class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but CrossWorlds recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 124 lists the deprecated methods for the BusObj class. If you have not used Map Designer before, ignore this section.

Table 124. *Deprecated methods, BusObj Class*

---

Former Method	Replacement
getCount()	BusObjArray.size()
getKeys()	keysToString()
getValues()	toString()
not	standard Java NOT operator, "!"

---

Table 124. *Deprecated methods, BusObj Class (continued)*

<b>Former Method</b>	<b>Replacement</b>
<code>set(BusObj inputBusObj)</code> All methods that took a child business object or child business object array as an input argument	<code>copy()</code> Get a handle to the child business object or business object array and use the methods of the <code>BusObj</code> or <code>BusObjArray</code> 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 11. BusObjArray class

The methods documented in this chapter operate on objects of the IBM 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.

**Note:** The `BusObjArray` class is used for both collaboration development and mapping; check the Notes section for each method's usage issues.

Table 125 lists the methods of the `BusObjArray` class.

*Table 125. BusObjArray method summary*

Method	Description	Page
<code>addElement()</code>	Add a business object to this business object array.	348
<code>duplicate()</code>	Create a business object array ( <code>BusObjArray</code> object) exactly like this one.	348
<code>elementAt()</code>	Retrieve a single business object by specifying its position in this business object array.	349
<code>equals()</code>	Compare another business object array with this one.	349
<code>getElements()</code>	Retrieve the contents of this business object array.	350
<code>getLastIndex()</code>	Retrieve the last available index from a business object array.	350
<code>max()</code>	Retrieve the maximum value for the specified attribute among all elements in this business object array.	350
<code>maxBusObjArray()</code>	Returns the business objects that have the maximum value for the specified attribute, as a business object array ( <code>BusObjArray</code> object).	351
<code>maxBusObjs()</code>	Returns the business objects that have the maximum value for the specified attribute, as an array of <code>BusObj</code> objects.	352
<code>min()</code>	Retrieve the minimum value for the specified attribute among the business objects in this array.	353
<code>minBusObjArray()</code>	Returns the business objects that have the minimum value for the specified attribute, as a <code>BusObjArray</code> object.	354
<code>minBusObjs()</code>	Returns the business objects that have the minimum value for the specified attribute, as an array of <code>BusObj</code> objects.	355
<code>removeAllElements()</code>	Remove all elements from this business object array.	356
<code>removeElement()</code>	Remove a business object element from a business object array.	356
<code>removeElementAt()</code>	Remove an element at a particular position in this business object array.	357

Table 125. *BusObjArray* method summary (continued)

Method	Description	Page
<code>setElementAt()</code>	Set the value of a business object in a business object array.	357
<code>size()</code>	Return the number of elements in this business object array.	358
<code>sum()</code>	Adds the values of the specified attribute for all business objects in this business object array.	358
<code>swap()</code>	Reverse the positions of two business objects in this 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.	358
<code>toString()</code>	Retrieve the values in this business object array as a single string.	359

**Note:** See “Exceptions and exception types” on page 328 for an important clarification on exception handling with this class. The section applies to exceptions in `BusObjArray` and `BusObj` only.

---

## 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 (`BusObjArray` object) exactly like this one.

### Syntax

```
BusObjArray duplicate()
```

### Return values

A business object array.

## Examples

The following example duplicates the `items` array, creating `newItems`.  
`BusObjArray newItems = items.clone();`

---

## 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 (int i=0, i<elements.length; 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

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, Integer, 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, Integer, 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`, `Integer`, `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`, `Integer`, `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, Integer, 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, Integer, 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()

Remove a business object 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*                      The element index.

### 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(5, new BusObj("Item"));
```

---

## size()

Return the number of elements in this business object array.

### Syntax

```
int size()
```

### Notes

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 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 `Integer`, `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 this 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 `swap()` call is the following array:

BusObjC	BusObjB	BusObjA
---------	---------	---------

---

## toString()

Retrieve the values in this business object array as 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 12. CwBidiEngine class

The CxBidiEngine class provides methods for transforming business objects and strings from one bidirectional format to the other.

Table 126 summarizes the methods in the CxBidiEngine class.

Table 126. CwBidiEngine method summary

Method	Description	Page
BiDiB0Transformation()	Transforms BusinessObject type business objects from one bidirectional format to the other format.	361
BiDiBusObjTransformation()	Transforms BusObj type business objects from one bidirectional format to the other format.	362
BiDiStringTransformation()	Transforms strings from one bidirectional format to the other.	363

---

### 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 127 on page 362 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 127 on page 362 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 363.

---

## 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 127 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 127 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 127. 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 128 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 128 for the valid values of this string. If this parameter is null, the method defaults to the standard Windows bidirectional format

Table 128. 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 13. 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 BaseDLM class. All maps are derived or subclassed from BaseDLM so they have access to `getDBConnection()`.

Table 126 summarizes the methods in the CwDBConnection class.

Table 129. CwDBConnection method summary

Method	Description	Page
<code>beginTransaction()</code>	Begins an explicit transaction for the current connection.	365
<code>commit()</code>	Commits the active transaction associated with the current connection.	366
<code>executeSQL()</code>	Executes a static SQL query by specifying its syntax and an optional parameter array.	368
<code>executePreparedSQL()</code>	Executes a prepared SQL query by specifying its syntax and an optional parameter array.	367
<code>executeStoredProcedure()</code>	Executes an SQL stored procedure by specifying its name and parameter array.	370
<code>getUpdateCount()</code>	Returns the number of rows affected by the last write operation to the database.	371
<code>hasMoreRows()</code>	Determines whether the query result has more rows to process.	371
<code>inTransaction()</code>	Determines whether a transaction is in progress in the current connection.	372
<code>isActive()</code>	Determines whether the current connection is active.	372
<code>nextRow()</code>	Retrieves the next row from the query result.	373
<code>release()</code>	Releases use of the current connection, returning it to its connection pool.	373
<code>rollback()</code>	Rolls back the active transaction associated with the current connection.	374

---

### 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.

If you do *not* use `beginTransaction()` to specify the beginning of the explicit transaction, the database executes each SQL statement as a separate transaction.

**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 `BaseDLM` 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

`commit()`, `getDBConnection()`, `inTransaction()`, `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 map. If the map is successful, ICS commits this database transaction. If the map is *not* successful, ICS implicitly rolls back the database transaction. Regardless of the success of the map, ICS logs a warning.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BasedLM` class. Make sure that this connection uses explicit transaction bracketing.

## Examples

For an example of committing a transaction, see the example for `beginTransaction()`.

## See also

`beginTransaction()`, `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 `executePreparedStatement()` 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 `executePreparedStatement()`. The `executePreparedStatement()` method saves this prepared statement in memory. Use `executePreparedStatement()` 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 `executePreparedStatement()`, 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.

## See also

`beginTransaction()`, `commit()`, `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 procedures with the limitation that this stored procedure *cannot* use any `OUT` parameters  
To execute stored procedures with `OUT` parameters, use the `executeStoredProcedure()` method.

## 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 relationship table, see

## See also

`executePreparedSQL()`, `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.

### 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

`executePreparedSQL()`, `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()`, `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

`CwDBSQLException` – 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

`executePreparedSQL()`, `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 the 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

`beginTransaction()`, `commit()`, `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

`getConnection()`, `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

`hasMoreRows()`, `executePreparedSQL()`, `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 map instance. 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 map instance implicitly releases it at the end of the current map 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 map. 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

`getDBConnection()`, `inTransaction()`, `isActive()`

---

## rollback()

Rolls back the active transaction associated with the current connection.

## Syntax

```
void rollback()
```

## Parameters

None.

## Return values

None.

## Exceptions

`CwDBConnectionException` – If a database error occurs.

## 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 map. If the map is successful, ICS commits this database transaction. If the map is *not* successful, ICS implicitly rolls back the database transaction. Regardless of the success of the map, ICS logs a warning.

Before beginning an explicit transaction, you must create a `CwDBConnection` object with the `getDBConnection()` method from the `BasedLM` class. Make sure that this connection uses explicit transaction bracketing.

## See also

`beginTransaction()`, `commit()`, `getDBConnection()`, `inTransaction()`



---

## Chapter 14. CwDBStoredProcedureParam class

A CwDBStoredProcedureParam object describes a single parameter for a stored procedure. Table 130 summarizes the methods in the CwDBStoredProcedureParam class.

Table 130. CwDBStoredProcedureParam method summary

Method	Description	Page
CwDBStoredProcedureParam()	Constructs a new instance of CwDBStoredProcedureParam that holds argument information for the parameter of a stored procedure.	377
getParamType()	Retrieves the in/out type of the current stored-procedure parameter as an integer constant.	378
getValue()	Retrieves the value of the current stored-procedure parameter.	379

---

### 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 131.
- 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.

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

`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 `getParamType()` 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 131 shows.

Table 131. 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

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 getValue() 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 15. DtpConnection class

The `DtpConnection` class is part of the Data Transformation Package (DTP). It provides methods for executing SQL queries on the relationship database. To instantiate this class, you must call `getRelConnection()` in the `BaseDLM` class. All maps are derived or subclassed from `BaseDLM` so they have access to `getRelConnection()`.

**Important:** The `DtpConnection` class and its methods are supported for backward compatibility *only*. These *deprecated methods* will not generate errors, but you should avoid using them and migrate existing code to the new methods. The deprecated methods might be removed in a future release. In new map development, use the `CwDBCConnection` class and its methods to establish a database connection.

Table 132 summarizes the methods in the `DtpConnection` class.

Table 132. *DtpConnection* method summary

Method	Description	Page
<code>beginTran()</code>	Begins an SQL transaction for the relationship database.	381
<code>commit()</code>	Commits the current transaction in the relationship database.	382
<code>executeSQL()</code>	Executes a SQL query in the relationship database by specifying a CALL statement.	383
<code>execStoredProcedure()</code>	Executes an SQL stored procedure in the relationship database by specifying its name and parameter array.	384
<code>getUpdateCount()</code>	Returns the number of rows affected by the last write operation to the relationship database.	385
<code>hasMoreRows()</code>	Determines whether the query result has more rows to process.	385
<code>inTransaction()</code>	Determines whether a transaction is in progress in the relationship database.	386
<code>nextRow()</code>	Retrieves the next row in the query result vector.	386
<code>rollback()</code>	Rolls back the current transaction in the relationship database.	387

---

### **beginTran()**

Begins an SQL transaction for the relationship database.

#### **Syntax**

```
void beginTran()
```

#### **Parameters**

None.

#### **Return values**

None.

## Exceptions

`DtpConnectionException` – If a database error occurs.

## Notes

The `beginTran()`, `commit()` and `rollback()` methods together provide transaction support for SQL queries.

Before beginning a transaction, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.

## Examples

The following example uses a transaction to execute a query for inserting rows into a table in the `SapCust` relationship.

```
DtpConnection connection = getRelConnection("SapCust");

// begin a transaction
connection.beginTran();

// insert a row
connection.executeSQL("insert...");

// commit the transaction
connection.commit();
```

## See also

`commit()`, `getRelConnection()`, `inTransaction()`, `rollback()`

---

## `commit()`

Commits the current transaction in the relationship database.

## Syntax

```
void commit()
```

## Parameters

None.

## Return values

None.

## Exceptions

`DtpConnectionException` – If a database error occurs.

## Notes

The `beginTran()`, `commit()` and `rollback()` methods together provide transaction support for SQL queries.

Before beginning a transaction, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.



## Examples

The following example uses a transaction to execute a query for inserting rows into a table in the SapCust relationship.

```
DtpConnection connection = getRelConnection("SapCust");

// begin a transaction
connection.beginTran();

// insert a row
connection.executeSQL("insert...");

// commit the transaction
connection.commit();
```

## See also

`beginTran()`, `getRelConnection()`, `inTransaction()`, `rollback()`

---

## executeSQL()

Executes a SQL query in the relationship database by specifying a CALL statement.

## Syntax

```
void executeSQL(String query)
void executeSQL(String query, Vector queryParameters)
```

## Parameters

*query*                    The SQL query to run in the relationship database.

*queryParameters*        A Vector object of arguments to pass to parameters in the SQL query.

## Return values

None.

## Exceptions

`DtpConnectionException` – If a database error occurs.

## Notes

Before executing a query with `executeSQL()`, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.

The SQL statements you can execute include `INSERT`, `SELECT`, `DELETE`, and `UPDATE`. You can also execute stored procedures with the limitation that this stored procedure *cannot* use any `OUT` parameters. To execute stored procedures with `OUT` parameters, use the `execStoredProcedure()` method.

## Examples

The following example executes a query for inserting rows into a table in the SapCust relationship.

```
DtpConnection connection = getRelConnection("SapCust");

// begin a transaction
connection.beginTran();
```

```
// insert a row
connection.executeSQL("insert...");

// commit the transaction
connection.commit();

// release the database connection
releaseRelConnection(true);
```

## See also

`execStoredProcedure()`, `getRelConnection()`, `hasMoreRows()`, `nextRow()`

---

## execStoredProcedure()

Executes an SQL stored procedure in the relationship database by specifying its name and parameter array.

### Syntax

```
void execStoredProcedure(String storedProcedure,
    Vector storedProcParameters)
```

### Parameters

*storedProcedure*

The name of the SQL stored procedure to run in the relationship database.

*storedProcParameters*

A Vector object of parameters to pass to the stored procedure. Each parameter is an instance of the `UserStoredProcedureParam` class.

### Return values

None.

### Exceptions

`DtpConnectionException` – If a database error occurs.

### Notes

Before executing a stored procedure with `execStoredProcedure()`, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.

You can also use the `executeSQL()` 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 `execStoredProcedure()` to execute it. Unlike with `executeSQL()`, you do not have to pass in the full SQL statement to execute the stored procedure. With `execStoredProcedure()`, you need to pass in only the name of the stored procedure and a Vector parameter array of `UserStoredProcedureParam` objects. The `execStoredProcedure()` method can determine the number of parameters from the *storedProcParameters* array and builds the calling statement for the stored procedure.

## See also

`executeSQL()`, `getRelConnection()`, `hasMoreRows()`, `nextRow()`

---

## getUpdateCount()

Returns the number of rows affected by the last write operation to the relationship database.

### Syntax

```
int getUpdateCount()
```

### Parameters

None.

### Return values

Returns an `int` representing the number of rows affected by the last write operation.

### Exceptions

`DtpConnectionException` – If a database error occurs.

### Notes

Before using this method, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.

This method is useful after you send an `UPDATE` or `INSERT` statement on the relationship database and you want to determine the number of rows that the `SQL` statement has affected.

### See also

`executeSQL()`, `getRelConnection()`

---

## 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

`DtpConnectionException` – If a database error occurs.

### Notes

The `hasMoreRows()` method determines whether the query associated with the current relationship database 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

`nextRow()`, `executeSQL()`, `getUpdateCount()`

---

## `inTransaction()`

Determines whether a transaction is in progress in the relationship database.

### Syntax

```
boolean inTransaction()
```

### Parameters

None.

### Return values

Returns "True" if a transaction is in progress.

### Exceptions

`DtpConnectionException` – If a database error occurs.

### Notes

Before beginning a transaction, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.

## See also

`beginTran()`, `commit()`, `getRelConnection()`, `rollback()`

---

## `nextRow()`

Retrieves the next row in the query result vector.

### Syntax

```
Vector nextRow()
```

### Parameters

None.

### Return values

Returns the next row of the query result as a `Vector` object.

### Exceptions

`DtpConnectionException` – If a database error occurs.

### Notes

The `nextRow()` method returns one row of data from the query associated with the current relationship database. 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 data from the initial query.

## See also

`hasMoreRows()`, `executeSQL()`, `getUpdateCount()`

---

## **rollback()**

Rolls back the current transaction in the relationship database.

## Syntax

```
void rollback()
```

## Parameters

None.

## Return values

None.

## Exceptions

`DtpConnectionException` – If a database error occurs.

## Notes

The `beginTran()`, `commit()` and `rollback()` methods together provide transaction support for SQL queries.

Before beginning a transaction, you must create a `DtpConnection` object with the `getRelConnection()` method from the `BaseDLM` class.

## See also

`beginTran()`, `commit()`, `getRelConnection()`, `inTransaction()`



---

## Chapter 16. DtpDataConversion class

One of the most common tasks in business object mapping is the conversion of attribute values from one data type to another, a process called *data conversion*. The `DtpDataConversion` class provides a simple way to perform data conversions.

The data type classes in the `java.lang` package contain some conversion methods, but all possible conversions are not supported. The `DtpDataConversion` class consolidates many data conversion methods into one class and it supports the most common conversions that you perform in maps. The `getType()` and `isOKToConvert()` methods make it easy to determine whether specific conversions are possible.

All methods in this class are declared as static. Table 133 summarizes the methods of the `DtpDataConversion` class.

Table 133. *DtpDataConversion* method summary

Method	Description	Page
<code>getType()</code>	Determines the data type of a value.	389
<code>isOKToConvert()</code>	Determines whether it is possible to convert a value from one data type to another.	390
<code>toBoolean()</code>	Converts a Java object to a <code>Boolean</code> object.	392
<code>toDouble()</code>	Converts an object or primitive data type to a <code>Double</code> object.	393
<code>toFloat()</code>	Converts an object or primitive data type to a <code>Float</code> object.	393
<code>toInteger()</code>	Converts an object or primitive data type to an <code>Integer</code> object.	394
<code>toPrimitiveBoolean()</code>	Converts a <code>String</code> or <code>Boolean</code> object to the primitive <code>boolean</code> data type.	395
<code>toPrimitiveDouble()</code>	Converts an object or primitive data type to the primitive <code>double</code> data type.	395
<code>toPrimitiveFloat()</code>	Converts an object or primitive data type to the primitive <code>float</code> data type.	396
<code>toPrimitiveInt()</code>	Converts an object or primitive data type to the primitive <code>int</code> data type.	397
<code>toString()</code>	Converts an object or primitive data type to a <code>String</code> object.	398

---

### getType()

Determines the data type of a value.

#### Syntax

```
int getType(Object objectData)
int getType(int integerData)
int getType(float floatData)
int getType(double doubleData)
int getType(boolean booleanData)
```

#### Parameters

*objectData* Any Java object.

<i>integerData</i>	Any primitive int variable.
<i>floatData</i>	Any primitive float variable.
<i>doubleData</i>	Any primitive double variable.
<i>booleanData</i>	Any primitive boolean variable.

## Return values

Returns an integer representing the data type of the parameter you pass. You can interpret the return value by comparing it to one of these constants which are declared as static and final in the `DtpDataConversion` class:

<i>INTEGER_TYPE</i>	The data is a primitive int value or Integer object.
<i>STRING_TYPE</i>	The data is a String object.
<i>FLOAT_TYPE</i>	The data is a primitive float value or Float object.
<i>DOUBLE_TYPE</i>	The data is a primitive double value or Double object.
<i>BOOL_TYPE</i>	The data is a primitive boolean value or Boolean object.
<i>DATE_TYPE</i>	The data is a Date object.
<i>LONGTEXT_TYPE</i>	The data is a LongText object.
<i>UNKNOWN_TYPE</i>	The data is of an unknown type.

## Exceptions

None.

## Notes

You can use the return values from `getType()` in the `OKToConvert()` method to determine whether a conversion is possible between two given data types.

## Examples

```
int conversionStatus = DtpDataConversion.isOKToConvert(
    DtpDataConversion.getType(srcObject),
    DtpDataConversion.getType(destObject));

switch(conversionStatus)
{
    case DtpDataConversion.OKTOCONVERT:
        // go ahead and convert
        break;
    case DtpDataConversion.POTENTIALDATALOSS:
        // convert, then check value
        break;
    case DtpDataConversion.CANNOTCONVERT:
        // return an error
        break;
}
```

## See also

`isOKToConvert()`

---

## isOKToConvert()

Determines whether it is possible to convert a value from one data type to another.



## Syntax

```
int isOKToConvert(int srcDatatype, int destDataType)  
int isOKToConvert(String srcDataTypeStr, String destDataTypeStr)
```

## Parameters

*srcDataType* Integer returned by `getType()`, which represents the data type of the source value that you want to convert.

*destDataType* Integer returned by `getType()`, which represents the data type to which you want to convert the source value.

*srcDataTypeStr* String containing the data type name for the source value that you want to convert. Possible values are: `Boolean`, `boolean`, `Double`, `double`, `Float`, `float`, `Integer`, `int`, and `String`.

*destDataTypeStr* String containing the data type name to which you want to convert the source value. Possible values are: `Boolean`, `boolean`, `Double`, `double`, `Float`, `float`, `Integer`, `int`, and `String`.

## Return values

Returns an integer specifying whether it is possible to convert a value of the source data type to a value of the destination data type. You can interpret the return value by comparing it to one of these constants, which are declared as `static` and `final` in the `DtpDataConversion` class:

`OKTOCONVERT` You can convert from the source to the destination data type.

`POTENTIALDATALOSS` You can convert, but there is a potential for data loss if the source value contains unconvertible characters or must be truncated to fit the destination data type.

`CANNOTCONVERT` The source data type cannot be converted to the destination data type.

## Exceptions

None.

## Notes

The `getType()` method returns an integer representing the data type of the value you pass as a parameter. You use the first form of `isOKToConvert()` together with `getType()` to determine whether a data conversion between two attributes is possible. In your `isOKToConvert()` method call, use `getType()` on both the source and destination attributes to generate the *srcDataType* and *destDataType* parameters.

The second form of the method accepts `String` values containing the data type names for the source and destination data. Use this form of the method if you know what the data types are, and you want to check whether you can perform a conversion.

Table 134 shows the possible conversions for each combination of source and destination data type. In the table:

- OK means you can convert the source type to the destination type with no data loss.
- DL means you can convert, but data loss might occur if the source contains unconvertible characters or must be truncated to fit the destination type.
- NO means you cannot convert the a value from source data type to the destination data type.

Table 134. Possible Conversions Between Data Types

SOURCE	D E S T I N A T I O N							
	int, Integer	String	float, Float	double, Double	boolean, Boolean	Date	Longtext	
int Integer	OK	OK	OK	OK	NO	NO	OK	
String	DL <sup>1</sup>	OK	DL <sup>1</sup>	DL <sup>1</sup>	DL <sup>2</sup>	DL	OK	
float, Float	DL <sup>3</sup>	OK	OK	OK	NO	NO	OK	
double, Double	DL <sup>3</sup>	OK	DL <sup>3</sup>	OK	NO	NO	OK	
boolean, Boolean	NO	OK	NO	NO	OK	NO	OK	
Date	NO	OK	NO	NO	NO	OK	OK	
Longtext	DL <sup>1</sup>	DL <sup>3</sup>	DL <sup>1</sup>	DL <sup>1</sup>	DL <sup>2</sup>	DL	OK	

<sup>1</sup>When converting a String or Longtext value to any numeric type, the String or Longtext value can contain only numbers and decimals. You must remove any other characters, such as currency symbols, from the String or Longtext value before converting. Otherwise, a `DtpIncompatibleFormatException` will be thrown.

<sup>2</sup>When converting a String or Longtext value to Boolean, the value of the String or Longtext should be "true" or "false". Any string that is not "true" (case does not matter) will be considered false.

<sup>3</sup>Because the source data type supports greater precision than the destination data type, the value might be truncated.

## Examples

```
if (DtpDataConversion.isOKToConvert(getType(mySource),
    getType(myDest)) == DtpDataConversion.OKTOCONVERT)
    // map these attributes
else
    // skip these attributes
```

## See also

`getType()`

## toBoolean()

Converts a Java object to a Boolean object.

## Syntax

```
Boolean toBoolean(Object objectData)
Boolean toBoolean(boolean booleanData)
```

## Parameters

- objectData* A Java object that you want to convert to Boolean. The only object currently supported is String.
- booleanData* Any primitive boolean variable.

## Return values

Returns a Boolean object.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to Boolean.

## Examples

```
Boolean MyBooleanObj = DtpDataConversion.toBoolean(MyStringObj);
```

## See also

`getType()`, `isOkToConvert()`, `toPrimitiveBoolean()`

---

## toDouble()

Converts an object or primitive data type to a Double object.

## Syntax

```
Double toDouble(Object objectData)
Double toDouble(int integerData)
Double toDouble(float floatData)
Double toDouble(double doubleData)
```

## Parameters

<i>objectData</i>	A Java object. The objects currently supported are: Float, Integer, and String.
<i>integerData</i>	Any primitive int variable.
<i>floatData</i>	Any primitive float variable.
<i>doubleData</i>	Any primitive double variable.

## Return values

Returns a Double object.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to Double.

## Examples

```
Double myDoubleObj = DtpDataConversion.toDouble(myInteger);
```

## See also

`getType()`, `isOkToConvert()`, `toPrimitiveDouble()`

---

## toFloat()

Converts an object or primitive data type to a Float object.

## Syntax

```
Float toFloat(Object objectData)  
Float toFloat(int integerData)  
Float toFloat(float floatData)  
Float toFloat(double doubleData)
```

## Parameters

*objectData* A Java object. The objects currently supported are: Double, Integer, and String.

*integerData* Any primitive int variable.

*floatData* Any primitive float variable.

*doubleData* Any primitive double variable.

## Return values

Returns a Float object.

## Exceptions

DtpIncompatibleFormatException – If the source data type cannot be converted to Float.

## Examples

```
Float myFloatObj = DtpDataConversion.toFloat(myInteger);
```

## See also

getType(), isOKToConvert(), toPrimitiveFloat()

---

## toInteger()

Converts an object or primitive data type to an Integer object.

## Syntax

```
Integer toInteger(Object objectData)  
Integer toInteger(int integerData)  
Integer toInteger(float floatData)  
Integer toInteger(double doubleData)
```

## Parameters

*objectData* A Java object. The objects currently supported are: Double, Float, and String.

*integerData* Any primitive int variable.

*floatData* Any primitive float variable.

*doubleData* Any primitive double variable.

## Return values

Returns an Integer object.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to `Integer`.

## Examples

```
Integer myIntegerObj = DtpDataConversion.toInteger(myFloat);
```

## See also

`getType()`, `isOkToConvert()`, `toPrimitiveInt()`

---

## toPrimitiveBoolean()

Converts a `String` or `Boolean` object to the primitive boolean data type.

### Syntax

```
boolean toPrimitiveBoolean(Object objectData)
```

### Parameters

*objectData*      A `String` or `Boolean` object that you want to convert to the primitive boolean data type.

### Return values

Returns a primitive boolean value.

### Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to `boolean`.

### Notes

This method can only handle `String` and `Boolean` objects.

### Examples

```
boolean MyBoolean = DtpDataConversion.toPrimitiveBoolean(MyStringObj);
```

### See also

`getType()`, `isOkToConvert()`, `toBoolean()`

For more information, see the Java Language Specification.

---

## toPrimitiveDouble()

Converts an object or primitive data type to the primitive double data type.

### Syntax

```
double toPrimitiveDouble(Object objectData)  
double toPrimitiveDouble(int integerData)  
double toPrimitiveDouble(float floatData)
```

## Parameters

<i>objectData</i>	A Java object. The objects currently supported are: Double, Float, Integer, and String.
<i>integerData</i>	Any primitive int variable.
<i>floatData</i>	Any primitive float variable.

## Return values

Returns a primitive double value.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to double.

## Notes

This method can only handle String, Integer, Float, and Double objects; and the return value may not equal the input value accurately.

The limitation of this method is the same as the limitation of the double type in Java.

The largest positive finite double literal is 1.79769313486231570e+308. The smallest positive finite nonzero literal of type double is 4.94065645841246544e-324, with 15 significant decimal digits (on order of 999,999,999,999.99, in the range of billions).

## Examples

```
double myDouble = DtpDataConversion.toPrimitiveDouble(myObject);
```

## See also

`getType()`, `isOKToConvert()`, `toDouble()`

For more information, see the Java Language Specification.

---

## toPrimitiveFloat()

Converts an object or primitive data type to the primitive float data type.

## Syntax

```
float toPrimitiveFloat(Object objectData)  
float toPrimitiveFloat(int integerData)  
float toPrimitiveFloat(double doubleData)
```

## Parameters

<i>objectData</i>	A Java object. The objects currently supported are: Double, Float, Integer, and String.
<i>integerData</i>	Any primitive int variable.
<i>doubleData</i>	Any primitive double variable.

## Return values

Returns a primitive float value.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to float.

## Notes

This method can only handle `String`, `Integer`, `Float`, and `Double` objects; and there will be some data loss when the input type is `Double`.

The limitation of this method is the same as the limitation of the float type in Java.

The largest positive finite float literal is `3.40282347e+38f`. The smallest positive finite nonzero literal of type float is `1.40239846e-45f`, with 6-7 significant figures. Values above `99,999.99` should not be used with this data type.

## Examples

```
float myFloat = DtpDataConversion.toPrimitiveFloat(myInteger);
```

## See also

`getType()`, `isOkToConvert()`, `toFloat()`

For more information, see the Java Language Specification.

---

## toPrimitiveInt()

Converts an object or primitive data type to the primitive int data type.

## Syntax

```
int toPrimitiveInteger(Object objectData)  
int toPrimitiveInteger(float floatData)  
int toPrimitiveInteger(double doubleData)
```

## Parameters

<i>objectData</i>	A Java object. The objects currently supported are: <code>Double</code> , <code>Float</code> , <code>Integer</code> , and <code>String</code> .
<i>floatData</i>	Any primitive float variable.
<i>doubleData</i>	Any primitive double variable.

## Return values

Returns a primitive int value.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to integer.

## Notes

This method can only handle String, Integer, Float, and Double objects; and there will be some data loss when the input type is Float or Double.

The limitation of this method is the same as the limitation of the int type in Java.

The largest positive hexadecimal and octal literals of type int are 0xffffffff and 017777777777, respectively, which equal 2147483647 (2<sup>31</sup>-1). The most negative hexadecimal and octal literals of type int are 0x80000000 and 020000000000, respectively, each of which represents the decimal value -2147483648 (-2<sup>31</sup>). The hexadecimal and octal literals 0xffffffff and 037777777777, respectively, represent the decimal value -1.

## Examples

```
int myInt = DtpDataConversion.toPrimitiveInt(myObject);
```

## See also

`getType()`, `isOkToConvert()`, `toInteger()`

For more information, see the Java Language Specification.

---

## toString()

Converts an object or primitive data type to a String object.

## Syntax

```
String toString(Object objectData)  
String toString(int integerData)  
String toString(float floatData)  
String toString(double doubleData)
```

## Parameters

<i>objectData</i>	A Java object. The objects currently supported are: Double, Float, and Integer.
<i>integerData</i>	Any primitive int variable.
<i>floatData</i>	Any primitive float variable.
<i>doubleData</i>	Any primitive double variable.

## Return values

Returns a String object.

## Exceptions

`DtpIncompatibleFormatException` – If the source data type cannot be converted to String.

## Examples

```
String myString = DtpDataConversion.toString(myObject);
```

## See also

`getType()`, `isOkToConvert()`



---

## Chapter 17. DtpDate class

The DtpDate class compares time and date values, sets their formats, and returns components of a time and date value.

The static (class) methods operate on the class name. The static methods take a set of business objects and return the earliest or latest dates or the business objects that contain the earliest or latest dates.

Instance methods operate on a date object. You pass a date value to the DtpDate constructor and you can then manipulate the resulting date object. Instance methods let you retrieve, format, and change the values associated with the date. You can also set the formats in which you want to handle dates.

The data conversion methods are useful when one application stores dates in one format and another application stores dates in another format. For example, SAP might send a date in the format 26/8/1999 15:23:20 but Clarify might need the date in the format August 26, 1999 15:23:20.

The values passed to the DtpDate class must follow these rules:

---

Day	A number from 1 to 30. If a separator between the month, year, and date is not present in the date-time string and the date is in a numeric format, single characters must be preceded by a zero (0), as in 01
Month	A number from 1 to 12, a name such as January or February, or an abbreviated (3 character) month name such as Jan or Feb. If a separator between the month, year, and date is not present in the date-time string and the date is in a numeric format, single characters must be preceded by a zero (0), as in 01.
Year	A 4-digit number.
Hour	A value in the range 01 to 23, representing 24-hour format. AM or PM designations are not allowed.
Minutes	A number in the range 01 to 59.
Seconds	A number in the range 01 to 59.

---

Table 135 summarizes the methods in the DtpDate class. Note that static and instance methods are separated in this table but are in alphabetical order in the chapter.

Table 135. DtpDate method summary

---

Method	Description	Page
<b>Constructor</b>		
DtpDate()	Parse the date according to the format specified.	401
<b>Static methods</b>		
getMaxDate()	From a list of business objects, return the latest date as a DtpDate object.	413
getMinDate()	From a list of business objects, return the earliest date as a DtpDate object.	415
getMaxDateB0()	From a list of business objects, return those that contain the latest date.	414
getMinDateB0()	From a list of business objects, return those that contain the earliest date.	417

---

Table 135. DtpDate method summary (continued)

Method	Description	Page
<b>Instance methods</b>		
addDays()	Add the specified number of days to this date.	402
addWeekdays()	Add the specified number of weekdays to this date.	403
addYears()	Add the specified number of years to this date.	404
after()	Check whether this date follows the date passed in as the input parameter.	405
before()	Check whether this date precedes the date passed in as the parameter.	406
calcDays()	Calculate the number of days between this date and another date.	406
calcWeekdays()	Calculate the number of weekdays between this date and another date.	407
get12MonthNames()	Return the current short-name representation of the twelve months for this date.	408
get12ShortMonthNames()	Return the current full-name representation of the twelve months for this date.	408
get7DayNames()	Return the current names for the seven days in the week for this date.	408
getCWDate()	Reformats this date into the IBM generic date format.	409
getDayOfMonth()	Return the day of the month for this date.	409
getDayOfWeek()	Return the day of the week for this date.	410
getHours()	Return the hours value for this date.	410
getIntDay()	Return the day of the week in this date as an integer.	410
getIntDayOfWeek()	Return the day of the week for this date.	411
getIntMilliseconds()	Return the milliseconds value from this date.	411
getIntMinutes()	Return the minutes value in this date as an integer.	411
getIntMonth()	Return the month in this date as an integer.	412
getIntSeconds()	Return the seconds in this date as an integer.	412
getIntYear()	Return the year in this date as an integer.	412
getMSSince1970()	Return the number of milliseconds between January 1, 1970 00:00:00 and this date.	413
getMinutes()	Return the minutes value from this date.	418
getMonth()	Return the full name representation of the month in this date.	418
getNumericMonth()	Return the month value from this date in numeric format.	418
getSeconds()	Return the seconds value from this date as a string.	419
getShortMonth()	Return the short name representation of the month name from this date.	419
getYear()	Return the year value in this date.	420
set12MonthNames()	Change the full-name representation for the twelve month names for this date.	420
set12MonthNamesToDefault()	Restore the full-name representation for the twelve month names to the default values for this date.	421
set12ShortMonthNames()	Change the short-name representation of the twelve month names for this date.	421
set12ShortMonthNamesToDefault()	Restore the short-name representation of the twelve month names to the default values for this date.	421

Table 135. DtpDate method summary (continued)

Method	Description	Page
set7DayNames()	Change the names of the seven days in the week for this date.	422
set7DayNamesToDefault()	Restore the names of the seven days in the week to the default values for this date.	422
toString()	Return the date in a specified format or the default format.	422

## DtpDate()

Parse the date according to the format specified.

### Syntax

```
public DtpDate()

public DtpDate(String dateTimeStr, String format)

public DtpDate(String dateTimeStr, String format, String[] monthNames,
                String[] shortMonthNames)

public DtpDate(long msSince1970, boolean isLocalTime)
```

### Parameters

*dateTimeStr* The date-time in the form of a string.

*format* The date format. See Notes for details.

*monthNames* An array of strings representing the full 12 month names. If null, the default value is January, February, March, and so on.

*shortMonthNames* An array of strings representing the short month name. If this is null but *monthNames* is not null, this value is the first 3 letters of the full month names, such as Jan, Feb, Mar, Apr, and so on.

*msSince1970* The number of milliseconds since January 1, 1970 00:00:00.

*isLocalTime* Set this to true if the time is already a local time, or to false otherwise.

### Return values

None

### Exceptions

DtpDateException - When the constructor encounters parsing errors. This may occur if the date is not in the specified format.

### Notes

The first form of the constructor does not take any parameters. It assigns the current date on the system to the new DtpDate object. It does not throw DtpDateException.

The second and the third forms of the constructor parse the date according to the specified date *format* and extract out the day, month, year, hour, minute, and second values. These can be retrieved and reformatted later with other `DtpDate` methods.

For example, a month can be retrieved in one of the following formats:

- The full-name representation (the default format): January, February, March, April, May, June, July, August, September, October, November, and December
- The numeric format: 1-12
- The short-name representation, which consists of the first three letters of each month name: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec

The retrieved data does not depend of the context of the other data.

You can change the full-name and short-name representations of the month in the following ways:

- With the `set12MonthNames()` and `set12ShortMonthNames()` methods respectively
- By passing the representation as a parameter into the third form of the `DtpDate()` constructor

The fourth form of the constructor takes the number of milliseconds since January 1, 1970 00:00:00. Many applications represent the date in this manner.

### Date format

In the date *format*, the date always precedes the time. The time is optional. If it is missing in a date-time string, the hours, minutes, and seconds have a default value of 00.

The date format uses the following case sensitive key letters:

---

D	day
M	month
Y	year
h	hours
m	minutes
s	seconds

---

These key letters may be separated by a separator such as `"/"` or `"-"`.

## Examples

The following examples show the `DtpDate()` constructor creating new date objects `aDate`, `date2`, and `date3`:

```
Dtpdate aDate = new DtpDate("5/21/1997 15:23:01", "M/D/Y h:m:s");
DtpDate date2 = new DtpDate("05211997 152301", "MDY hms");
DtpDate date3 = new DtpDate("Jan 10, 1999 10:00:00", "M D, Y h:m:s");
```

The following date format results in the `DtpDateException` being thrown:

```
h:m:s D/M/Y
```

---

## addDays()

Add the specified number of days to this date.

## Syntax

```
public DtpDate addDays(int numberOfDays)
```

## Parameters

*numberOfDays* An integer number. If it is a negative number, the new date will be the date *numberOfDays* days before the current instance of `DtpDate`.

## Return values

A new `DtpDate` object.

## Exceptions

`DtpDateException`

## Notes

The `addDays()` method adds the specified number of days to this date. You can use the `get()` methods to retrieve information about the resulting new date. The `DtpDate` object returned inherits all the properties of the current object of `DtpDate`, such as month names, date format, and so on.

The new date will be adjusted to be a valid date. For example, adding five days to January 29, 1999 00:00:00 results in February 03, 1999 00:00:00, and adding -30 days results in December 30, 1998 00:00:00.

Adding days does not affect the time of day.

## Examples

```
try
{
    DtpDate today = new DtpDate();
    DtpDate tomorrow = today.addDays(1);
    System.out.println("Tomorrow is "
        + tomorrow.getDayOfMonth() + "/"
        + tomorrow.getNumericMonth() + "/"
        + tomorrow.getYear() + " "
        + tomorrow.getHours() + ":"
        + tomorrow.getMinutes() + ":"
        + tomorrow.getSeconds());
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

## See also

`addWeekdays()`, `addYears()`

---

## addWeekdays()

Add the specified number of weekdays to this date.

## Syntax

```
public DtpDate addWeekdays(int numberOfWeekdays)
```

## Parameters

*numberOfWeekdays*

An integer number. If it is a negative number, the new date will be the date that is *numberOfWeekdays* weekdays before the date represented by the current `DtpDate` object.

## Return values

A new `DtpDate` object.

## Exceptions

`DtpDateException`

## Notes

The `addWeekdays()` method adds the specified number of weekdays to this date. You can then use the `get` methods to retrieve the information of the resulting new date. The `DtpDate` returned will inherit all the properties of the current instance of `DtpDate`, such as month names, date format, and so on.

Only Monday, Tuesday, Wednesday, Thursday, and Friday, or the equivalent values, are considered to be weekdays. Monday is considered to be the first day of the week.

## Examples

```
try
{
    DtpDate today = new DtpDate("8/2/1999 00:00:00", "M/D/Y h:m:s");
    DtpDate fiveWeekdaysLater = today.addWeekdays(5);
    // The new date should be 8/9/1999 00:00:00
    System.out.println("Next month is "
        + fiveWeekdaysLater.getDayOfMonth() + "/"
        + fiveWeekdaysLater.getNumericMonth() + "/"
        + fiveWeekdaysLater.getYear() + " "
        + fiveWeekdaysLater.getHours() + ":"
        + fiveWeekdaysLater.getMinutes() + ":"
        + fiveWeekdaysLater.getSeconds());
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

## See also

`addDays()`, `addYears()`

---

## addYears()

Add the specified number of years to this date.

## Syntax

```
public DtpDate addYears(int numberOfYears)
```

## Parameters

*numberOfYears* An integer number. If it is a negative number, the new date will be the date that is *numberOfYears* years before the current `DtpDate` object.

## Return values

A new `DtpDate` object.

## Notes

The `addYears()` method adds the specified number of years to this date. You can then use the `get()` methods to retrieve the information of the resulting new date. The `DtpDate` returned inherits all the properties of the current instance of `DtpDate`, such as month names, date format, and so on.

## Examples

```
DtpDate today = new DtpDate();
DtpDate lastYear= today.addYears(-1);
System.out.println("Next month is "
    + lastYear.getDayOfMonth() + "/"
    + lastYear.getNumericMonth() + "/"
    + lastYear.getYear() + " "
    + lastYear.getHours() + ":"
    + lastYear.getMinutes() + ":"
    + lastYear.getSeconds());
```

## See also

`addDays()`, `addWeekdays()`

---

## after()

Check whether this date follows the date passed in as the input parameter.

## Syntax

```
public boolean after(DtpDate date)
```

## Parameters

*date* The date to compare with this date.

## Return values

Return `true` if this date follows the date passed in, and `false` if this date precedes the data passed in.

## Exceptions

`DtpDateException`

## Examples

```
try
{
    DtpDate today = new DtpDate();
    DtpDate tomorrow = yesterday.addDays(-1);
    // isAfter should be false.
    boolean isAfter = yesterday.after(today)
}
```

```
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

## See also

`before()`

---

## `before()`

Check whether this date precedes the date passed in as the parameter.

## Syntax

```
public boolean before(DtpDate date)
```

## Parameters

*date*                    The date to compare with this date.

## Return values

Return true if this date precedes the date passed in, and false if this date follows the data passed in.

## Exceptions

DtpDateException

## Examples

```
try
{
    DtpDate today = new DtpDate();
    DtpDate tomorrow = yesterday.addDays(-1);
    // isBefore should be true.
    boolean isBefore = yesterday.before(today)
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

## See also

`after()`

---

## `calcDays()`

Calculate the number of days between this date and another date.

## Syntax

```
public int calcDays(DtpDate date)
```

## Parameters

*date*                    The date to compare with this date.



## Return values

An int representing the number of days. This is always a positive number.

## Exceptions

DtpDateException

## Notes

The `calcDays()` method calculates the difference in the number of days between this date and another date. The result is always a whole number of days.

The difference between 19990615 00:30:59 and 19990615 23:59:59 is 0 days, and the difference between 19990615 23:59:59 and 19990616 00:01:01 is 1 day.

## Examples

```
try
{
    DtpDate today = new DtpDate();
    DtpDate tomorrow = today.addDays(1);
    int days = today.calcDays(tomorrow);
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

## See also

`calcWeekdays()`

---

## calcWeekdays()

Calculate the number of weekdays between this date and another date.

## Syntax

```
public int calcWeekdays(DtpDate date)
```

## Parameters

*date*                      The date to compare with this date.

## Return values

An int representing the number of weekdays. This is always a positive number.

## Exceptions

DtpDateException

## Notes

The `calcWeekdays()` method calculates the number of weekdays between this date and another date. The difference between Friday and Saturday is 0, and between Friday and Monday is 1. Weekdays are assumed to be Monday through Friday or the equivalent values. A weekday is not the same as a business day, since a holiday can fall on a weekday.

## Examples

```
try
{
    DtpDate toDay = new DtpDate();
    DtpDate tomorrow = toDay.addDays(1);
    int days = today.calcWeekdays(tomorrow);
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

## See also

`calcDays()`

---

## get12MonthNames()

Return the current full-name representation of the twelve months for this date.

## Syntax

```
public String[ ] get12MonthNames()
```

## Return values

An array of String objects containing the effective names of the twelve months.

## Examples

```
DtpDate toDay = new DtpDate();
String[] toDay.get12MonthNames();
```

## See also

`set12MonthNames()`, `set12MonthNamesToDefault()`

---

## get12ShortMonthNames()

Return the current short-name representation of the twelve months for this date.

## Syntax

```
public String[ ] get12ShortMonthNames()
```

## Return values

An array of String objects containing the effective short names of the twelve months.

## Examples

```
DtpDate toDay = new DtpDate();
String[] toDay.get12ShortMonthNames();
```

## See also

`set12ShortMonthNames()`, `set12ShortMonthNamesToDefault()`

---

## get7DayNames()

Return the current names for the seven days in the week for this date.

## Syntax

```
public String[] get7DayNames()
```

## Return values

An array of String objects containing the effective names for the seven days of the week.

## Examples

```
DtpDate toDay = new DtpDate();  
String[] toDay.get7DayNames();
```

## See also

```
set7DayNames(), set7DayNamesToDefault()
```

---

## getCWDate()

Reformats this date into the IBM generic date format.

## Syntax

```
public String getCWDate()
```

## Return values

A string representing the date in the IBM WebSphere InterChange Server generic business object format. The format is YMD hms. Examples of this format are:

- 19990615 150701
- 19990831 114122

## Notes

The IBM generic date format takes the form:

```
YYYYMMDD HHMMSS
```

## Examples

```
DtpDate toDay = new DtpDate();  
String genericDate = toDay.getCWDate();
```

---

## getDayOfMonth()

Return the day of the month for this date.

## Syntax

```
public String getDayOfMonth()
```

## Return values

The string representing the day of the month, such as 01, 20, 30, and so on.

## Examples

```
DtpDate toDay = new DtpDate();  
String dayOfMonth = toDay.getDayOfMonth();
```

## See also

`getIntDay()`

---

## `getDayOfWeek()`

Return the day of the week for this date.

### Syntax

```
public String getDayOfWeek()
```

### Return values

A string indicating day of the week, such as Monday, Tuesday, and so on.

### Examples

```
DtpDate toDay = new DtpDate();  
String dayOfWeek = toDay.getDayOfWeek();
```

## See also

`getIntDayOfWeek()`

---

## `getHours()`

Return the hours value for this date.

### Syntax

```
public String getHours()
```

### Return values

The string representing the hour value which will be between 00 and 23.

### Examples

```
DtpDate toDay = new DtpDate();  
String hours = toDay.getHours();
```

---

## `getIntDay()`

Return the day of the month in this date as an integer.

### Syntax

```
public int getIntDay()
```

### Return values

An int value which is the day of the month.

### Examples

```
DtpDate toDay = new DtpDate();  
int day = toDay.getIntDay();
```

## See also

`getDayOfMonth()`

---

## getIntDayOfWeek()

Return the day of the week in this date as an integer.

### Syntax

```
public int getIntDayOfWeek()
```

### Return values

An int value which is the day of the week. The possible values are 0 (Monday), 1 (Tuesday), 2 (Wednesday), 3 (Thursday), 4 (Friday), 5 (Saturday), or 6 (Sunday).

### Examples

```
DtpDate toDay = new DtpDate();  
int dayOfWeek = toDay.getIntDayOfWeek();
```

### See also

```
getDayOfWeek()
```

---

## getIntMilliseconds()

Return the milliseconds value from the date.

### Syntax

```
public int getIntMilliseconds()
```

### Return values

An int value which is the milliseconds. The range is 0-999.

### Examples

```
DtpDate toDay = new DtpDate();  
int millisecs = toDay.getIntMilliseconds();
```

---

## getIntMinutes()

Return the minutes value in this date as an integer.

### Syntax

```
public int getIntMinutes()
```

### Return values

An int value which is the minutes. The range is 0-59.

### Examples

```
DtpDate toDay = new DtpDate();  
int mins = toDay.getIntMinutes();
```

### See also

```
getMinutes()
```

---

## getIntMonth()

Return the month in this date as an integer.

### Syntax

```
public int getIntMonth()
```

### Return values

An int value which is the month. The range is 1 (January) - 12 (December).

### Examples

```
DtpDate toDay = new DtpDate();  
int month = toDay.getIntMonth();
```

### See also

`getMonth()`, `getNumericMonth()`

---

## getIntSeconds()

Return the seconds in this date as an integer.

### Syntax

```
public int getIntSeconds()
```

### Return values

An int value which is the seconds. The range is 0-59.

### Examples

```
DtpDate toDay = new DtpDate();  
int secs = toDay.getIntSeconds();
```

### See also

`getSeconds()`, `getMSSince1970()`

---

## getIntYear()

Return the year in this date as an integer.

### Syntax

```
public int getIntYear()
```

### Return values

An int value which is the year.

### Examples

```
DtpDate toDay = new DtpDate();  
int year = toDay.getIntYear();
```

### See also

`getYear()`

---

## getMSSince1970()

Return the number of milliseconds between January 1, 1970 00:00:00 and this date.

### Syntax

```
public long getMSSince1970()
```

### Return values

An integer number. It may be negative if this date is before January 1, 1970 00:00:00.

### Exceptions

DtpDateException

### Examples

```
try
{
    DtpDate toDay = new DtpDate();
    long ms = toDay.getMSSince1970();
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```

### See also

getSeconds()

---

## getMaxDate()

From a list of business objects, return the latest date as a DtpDate object.

### Syntax

```
public static DtpDate getMaxDate(BusObjArray boList, String attr,
    String dateFormat)
```

### Parameters

<i>boList</i>	A list of business objects.
<i>attr</i>	The attribute of the business object to use when doing the comparison. The attribute must be of type Date.
<i>dateFormat</i>	This is the date format. See DtpDate() for more details. If this is null, it is assumed that the date is the number of milliseconds since 1970.

### Return values

A DtpDate object that contains the max date.

### Exceptions

DtpIncompatibleB0TypeException - When the business objects in the list are not the same business object type.

DtpUnknownAttributeException - When the specified attribute is not a valid attribute in the business objects passed in.

DtpUnsupportedAttributeTypeException - When the type of the specified attribute is not one of the supported attribute types listed above.

All of these exceptions are subclasses of RunTimeEntityException.

## Notes

The getMaxDate() method scans through the list of business objects looking for the business object with the latest date, and returns that date in the form of a DtpDate object.

**Tip:** This method is a static method.

In the date evaluation, Jan 1, 2004 000000 is later than Jan 1, 2002 000000, which is later than Jan 1, 1999 000000

The date information is assumed to be stored in the attribute name passed into the method. If an object has null date information, it is ignored. If all of the objects have null date information, null is returned.

## Examples

```
try
{
    DtpDate maxDate = DtpDate.getMaxDate(bos, "Start Date",
        "D/M/Y h:m:s");
}
catch ( RunTimeEntityException err )
{
    System.out.println(err.getMessage());
}
```

## See also

getMinDate(), getMaxDateBO()

---

## getMaxDateBO()

From a list of business objects, return those that contain the latest date.

## Syntax

```
public static BusObj[] getMaxDateBO(BusObj[] boList, String attr,
    String dateFormat)
public static BusObj[] getMaxDateBO(BusObjArray boList, String attr,
    String dateFormat)
```

## Parameters

*boList* A list of business objects. It can be either an array of BusObj or an instance of BusObjArray. These business objects must be of the same business object type.

*attr* The attribute of the business object to compare with. The attribute must be of type Date.



*dateFormat* This is the date format. See `DtpDate()` for more details. If this is null, it is assumed that the date is the number of milliseconds since 1970.

## Return values

An array of business objects that have the latest date.

## Exceptions

All of these three exceptions are subclasses of `RunTimeEntityException`.

`DtpIncompatibleB0TypeException` - When the business objects in the list are not the same business object type.

`DtpUnknownAttributeException` - When the specified attribute is not a valid attribute in the business objects passed in.

`DtpUnsupportedAttributeTypeException` - When the type of the specified attribute is not one of the supported attribute types listed above.

`DtpDateException` - When the date format is invalid.

## Notes

The `getMaxDateB0()` method scans through the list of business objects looking for the business object with the latest date and returns that business object. If multiple business objects have the same max date, all objects with that date are returned.

**Tip:** This method is a static method.

In the evaluation of which date is earliest, Jan 1, 2004 000000 is later than Jan 1, 2002 000000, which is later than Jan 1, 1999 000000.

The date information is assumed to be stored in the attribute name passed into the method. If an object has null date information, that object is ignored. If all of the objects have null date information, null is returned.

## Examples

```
try
{
    BusObj[] max = DtpDate.getMaxDateB0(bos, "Start Date",
        "D/M/Y h:m:s");
}
catch ( RunTimeEntityException err )
{
    System.out.println(err.getMessage());
}
```

## See also

`getMaxDate()`, `getMinDateB0()`

---

## getMinDate()

From a list of business objects, return the earliest date as a `DtpDate` object.

## Syntax

```
public static DtpDate getMinDate(BusObjArray boList, String attr,
    String dateFormat)
```

## Parameters

<i>boList</i>	A list of business objects.
<i>attr</i>	The attribute of the business object to use when doing the comparison. The attribute must be of type Date.
<i>dateFormat</i>	The date format. See DtpDate() for more details. If this is null, it is assumed that the date is the number of milliseconds since 1970.

## Return values

A DtpDate object which contains the earliest date.

## Exceptions

DtpIncompatibleB0TypeException - When the business objects in the list are not the same business object type.

DtpUnknownAttributeException - When the specified attribute is not a valid attribute in the business objects passed in.

DtpUnsupportedAttributeTypeException - When the type of the specified attribute is not one of the supported attribute types listed above.

All of these exceptions are subclasses of RunTimeEntityException.

## Notes

The getMinDate() method scans through the list of business objects looking for the business object with the earliest date, and return that date in the form of a DtpDate object.

**Tip:** This method is a static method.

In the evaluation of dates, Jan 1, 1999 000000 is earlier than Jan 1, 2002 000000, which is earlier than Jan 1, 2004 000000.

The date information is assumed to be stored in the attribute name passed into the method. If an object has null date information, it is ignored. If all objects have null date information, null is returned.

## Examples

```
try
{
    DtpDate minDate = DtpDate.getMinDate(bos, "Start Date",
        "D/M/Y h:m:s");
}
catch ( RunTimeEntityException err )
{
    System.out.println(err.getMessage());
}
```

## See also

getMaxDate(), getMinDateB0()

---

## getMinDateBO()

From a list of business objects, return those that contain the earliest date.

### Syntax

```
public static BusObj[] getMinDateBO(BusObj[] boList, String attr,  
    String dateFormat)  
public static BusObj[] getMinDateBO(BusObjArray boList, String attr,  
    String dateFormat)
```

### Parameters

<i>boList</i>	A list of business objects.
<i>attr</i>	The attribute of the business object to use when doing the comparison. The attribute must be of type Date.
<i>dateFormat</i>	The date format. See DtpDate() for more details. If this is null, it is assumed that the date is the number of milliseconds since 1970.

### Return values

An array of business objects that have the date.

### Exceptions

DtpIncompatibleBOTypeException - When the business objects in the list are not the same business object type.

DtpUnknownAttributeException - When the specified attribute is not a valid attribute in the business objects passed in.

DtpUnsupportedAttributeTypeException - When the type of the specified attribute is not one of the supported attribute types listed above.

DtpDateException - When the date format is invalid.

All of these exceptions are subclass of RunTimeEntityException.

### Notes

The getMinDateBO() method scans through the list of business objects looking for the business object with the earliest date and returns that date in the form of a DtpDate object.

**Tip:** This method is a static method.

In the evaluation of the earliest date, Jan 1, 2004 000000 is later than Jan 1, 2002 000000 which is later than Jan 1, 1999 000000.

The date information is assumed to be stored in the attribute name passed into the method. If an object has null date information, it is ignored. If all of the objects have null date information, null is returned.

### Examples

```
try  
{  
    BusObj[] min = DtpDate.getMinDateBO(bos, "Start Date",  
        "D/M/Y h:m:s");
```

```
    }  
    catch ( RuntimeException err )  
    {  
        System.out.println(err.getMessage());  
    }  
}
```

## See also

`getMinDate()`, `getMaxDateB0()`

---

## getMinutes()

Return the minutes value from this date.

### Syntax

```
public String getMinutes()
```

### Return values

The string representing the minutes. The return value is between 00 and 59.

## See also

`getIntMinutes()`

---

## getMonth()

Return the full name representation of the month in this date.

### Syntax

```
public String getMonth()
```

### Return values

The name of the month, such as January, February, and so on.

## See also

`getIntMonth()`, `getNumericMonth()`, `getShortMonth()`

---

## getNumericMonth()

Return the month value from this date in numeric format.

### Syntax

```
public String getNumericMonth()
```

### Return values

The string in the numeric form for the month, such as 01, 02, and so on.

### Examples

```
DtpDate today = new DtpDate();  
System.out.println("Today is "  
    + today.getDayOfMonth() + "/"  
    + today.getNumericMonth() + "/"
```

```
+ toDay.getYear() + " "  
+ toDay.getHours() + ":"  
+ toDay.getMinutes() + ":"  
+ toDay.getSeconds());
```

## See also

`getIntMonth()`, `getMonth()`

---

## getSeconds()

Return the seconds value from this date as a string.

## Syntax

```
public String getSeconds()
```

## Return values

The string representing the seconds. The return value is between 00 and 59.

## Examples

```
DtpDate toDay = new DtpDate();  
System.out.println("Today is "  
+ toDay.getDayOfMonth() + "/"  
+ toDay.getNumericMonth() + "/"  
+ toDay.getYear() + " "  
+ toDay.getHours() + ":"  
+ toDay.getMinutes() + ":"  
+ toDay.getSeconds());
```

## See also

`getIntSeconds()`

---

## getShortMonth()

Return the short name representation of the month name from this date.

## Syntax

```
public String getShortMonth()
```

## Return values

The name of the month in the short format, such as Jan, Feb, and so on.

## Examples

```
DtpDate toDay = new DtpDate();  
DtpDate lastYear= toDay.addYears(-1);  
System.out.println("Next month is "  
+ lastYear.getShortMonth() + " "  
+ lastYear.getDayOfMonth() + ", "  
+ lastYear.getYear() + " "  
+ lastYear.getHours() + ":"  
+ lastYear.getMinutes() + ":"  
+ lastYear.getSeconds());
```

## See also

`getMonth()`, `set12ShortMonthNames()`, `set12ShortMonthNamesToDefault()`

---

## getYear()

Return the year value in this date.

### Syntax

```
public String getYear()
```

### Return values

The string representing the year. The year value includes the century. Examples are 1998 and 2004.

### Examples

```
DtpDate toDay = new DtpDate();
DtpDate lastYear= toDay.addYears(-1);
System.out.println("Next month is "
    + lastYear.getDayOfMonth() + "/"
    + lastYear.getNumericMonth() + "/"
    + lastYear.getYear() + " "
    + lastYear.getHours() + ":"
    + lastYear.getMinutes() + ":"
    + lastYear.getSeconds());
```

### See also

```
getIntYear()
```

---

## set12MonthNames()

Change the full-name representation for the twelve month names for this date.

### Syntax

```
public void set12MonthNames(String[] monthNames,
    boolean resetShortMonth)
```

### Parameters

*monthNames* An array of String containing the twelve month names. The first element is the first month of the year and the last element is the last month of the year.

*resetShortMonthNames*

By default, the short month names are the first three characters of the full month names. If this flag is set to true, the short month names will change based on the new full month names. If it is set to false, this method will not change the short month names.

### Return values

None.

### Exceptions

DtpDateException - When the month names passed in are not exactly 12 names.

### See also

```
get12MonthNames(), set12MonthNamesToDefault()
```

---

## set12MonthNamesToDefault()

Restore the full-name representation for the twelve month names to the default values for this date.

### Syntax

```
public void set12MonthNamesToDefault()
```

### Return values

None.

### Notes

The default names are January, February, March, and so on.

### See also

`get12MonthNames()`, `set12MonthNames()`

---

## set12ShortMonthNames()

Change the short-name representation of the twelve month names for this date.

### Syntax

```
public void set12ShortMonthNames(String[] shortMonths)
```

### Parameters

*shortMonths*     A list of business objects.

### Return values

None.

### Exceptions

`DtpDateException` - When the month names passed in are not exactly 12 names.

### See also

`get12ShortMonthNames()`, `set12ShortMonthNamesToDefault()`

---

## set12ShortMonthNamesToDefault()

Restore the short-name representation of the twelve month names to the default values for this date.

### Syntax

```
public void set12ShortMonthNamesToDefault()
```

### Return values

None

### Notes

The short month names are Jan, Feb, Mar, and so on.

## See also

`get12ShortMonthNames()`, `set12ShortMonthNames()`

---

## set7DayNames()

Change the names of the seven days in the week for this date.

### Syntax

```
public void set7DayNames(String[] dayNames)
```

### Parameters

*dayNames* An array of strings containing the seven days in a week. The first element should be the equivalent of Monday.

### Return values

None.

### Exceptions

`DtpDateException` - When exactly seven days are not specified.

## See also

`get7DayNames()`, `set7DayNamesToDefault()`

---

## set7DayNamesToDefault()

Restore the names of the seven days in the week to the default values for this date.

### Syntax

```
public void set7DayNamesToDefault()
```

### Return values

None.

### Notes

The default names are Monday, Tuesday, Wednesday, and so on.

## See also

`get7DayNames()`, `set7DayNames()`

---

## toString()

Return the date in a specified format or the default format.

### Syntax

```
public String toString()  
public String toString(String format)  
public String toString(String format boolean twelveHr)
```



## Parameters

<i>format</i>	The date format. See <code>DtpDate()</code> for more details.
<i>twelveHr</i>	A boolean that, if set to true, specifies that the method returns 12-hour time instead of 24-hour time.

## Return values

A string containing the date information, such as:  
19990930 053029 PM

Regardless of the format of the month position, the output string is always a 2 character integer representation (that is, 01 for January, 12 for December, and so forth).

## Exceptions

`DtpDateException` - When the date format is invalid.

## Examples

```
try
{
    DtpDate toDay = new DtpDate();
    String date = toDay.toString("Y/M/D h:m:s");
}
catch ( DtpDateException date_e )
{
    System.out.println(date_e.getMessage());
}
```



---

## Chapter 18. DtpMapService class

A submap is a map that you call from within another map. The `DtpMapService` class provides a method for running submaps. Table 136 summarizes the method in the `DtpMapService` class.

Table 136. *DtpMapService* method summary

Method	Description	Page
<code>runMap()</code>	Runs the map you specify.	425

---

### `runMap()`

Runs the map you specify.

#### Syntax

```
BusObj[] runMap(String mapName, String mapType,
                BusObj[] srcBOS, cwExecCtx)
```

#### Parameters

<i>mapName</i>	The name of the map to run.
<i>mapType</i>	The type of the map to run. Use the following constant <i>only</i> , which is defined in the <code>DtpMapService</code> class: <code>CWMAPTYPE</code> – an IBM WebSphere InterChange Server map
<i>srcBOS</i>	An array of business objects that are the source business objects for <i>mapName</i> .
<i>cwExecCtx</i>	A variable that contains the execution context for the current map. This variable is defined in the code that Map Designer generates for every map.

#### Return values

Returns an array of business objects that are the destination business objects of *mapName*.

#### Exceptions

- `MapFailureException` – If an error occurs while attempting to run *mapName*.
- `MapNotFoundException` – If *mapName* is not found in the repository.
- `CxMissingIDException` – See `maintainSimpleIdentityRelationship()`.

#### Notes

Use the `runMap()` method to call a submap from within another map. For more information on calling submaps, see “Transforming with a submap” on page 43.

#### Examples

The following code calls a submap to map an application-specific Address business object to the generic Address business object:

```
// Create the BusObj Array
BusObj[] rSrcB0s = new BusObj[1];
rSrcB0s[0] = MyCustomerObj.MyAddressObj[0];

// Make the call to the map service
OutObjName = DtpMapService.runMap(MyAppAddressToGenAddress,
    DtpMapService.CWMAPTYPE,rSrcB0s,cwExecCtx);
```

## See also

“Transforming with a submap” on page 43

---

## Chapter 19. DtpSplitString class

The `DtpSplitString` class provides a way to split or parse a string into tokens and retrieve the results. This class is useful for manipulating formatted strings such as composite keys, dates, or telephone numbers.

`DtpSplitString` is similar to the `StringTokenizer` class in the `java.util` package. However, when working with IBM WebSphere InterChange Server maps, `DtpSplitString` provides these advantages over `StringTokenizer`:

- The tokens in a `DtpSplitString` object are indexed. This makes it easy to extract the specific tokens you are interested in. For example, if you parse a telephone number (such as 650-555-1111) into three tokens using the dash (-) as a delimiter, you can extract the area code by referencing element 0 and build the rest of the telephone number by concatenating element 1 and element 2.
- A `DtpSplitString` object allows bidirectional scrolling of the tokens. As you navigate the elements using `nextElement()` and `prevElement()` all the elements remain available.

Table 137 summarizes the methods in the `DtpSplitString` class.

Table 137. *DtpSplitString* method summary

Method	Description	Page
<code>DtpSplitString()</code>	Constructs a new instance of <code>DtpSplitString</code> and parses a string into tokens.	427
<code>elementAt()</code>	Returns an element in the <code>DtpSplitString</code> object at the position you specify.	428
<code>firstElement()</code>	Returns the element in the <code>DtpSplitString</code> object at position zero.	428
<code>getElementCount()</code>	Returns an integer containing the total number of elements.	429
<code>getEnumeration()</code>	Returns an Enumeration of <code>String</code> objects where each <code>String</code> is one of the parsed tokens.	430
<code>lastElement()</code>	Returns the last element in the <code>DtpSplitString</code> object.	430
<code>nextElement()</code>	Returns the next element in the <code>DtpSplitString</code> object.	430
<code>prevElement()</code>	Returns the previous element in the <code>DtpSplitString</code> object.	431
<code>reset()</code>	Resets the current position number in the <code>DtpSplitString</code> object to zero.	432

---

### **DtpSplitString()**

Constructs a new instance of `DtpSplitString` and parses a string into tokens.

#### **Syntax**

```
DtpSplitString(String str, String delimiters)
```

#### **Parameters**

*str*                      The string to parse.

*delimiters* A String containing the delimiters used in *str*. There can be more than one delimiter, but each delimiter can be only one character in length.

## Notes

`DtpSplitString()` parses *str* into tokens, called elements, based on the specified delimiters. After calling `DtpSplitString()`, you can call any of the `DtpSplitString` class methods to select and retrieve specific elements.

## Examples

```
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");
```

---

## elementAt()

Returns an element in the `DtpSplitString` object at the position you specify.

## Syntax

```
String elementAt(int nth)
```

## Parameters

*nth* The position of the element to extract from the `DtpSplitString` object. The position of the first element is zero.

## Return values

Returns a String containing the element at the *nth* position.

## Exceptions

`DtpNoElementAtPositionException` – If you specify an invalid position for *nth*.

## Notes

Elements are numbered from first to last beginning with zero. For example, if the delimiters are commas and spaces, then the element at position two in the string, "This,is a test" is "a".

The `elementAt()` method returns the element at the specified position but does not change the current element position.

## Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");

//This call returns "a"
public String MyString.elementAt(2);
```

## See also

```
getElementCount()
```

---

## firstElement()

Returns the element in the `DtpSplitString` object at position zero.

## Syntax

```
String firstElement()
```

## Return values

Returns a String containing the element at position zero.

## Exceptions

DtpNoElementAtPositionException – If there are no elements.

## Notes

Elements in the DtpSplitString object are numbered from first to last beginning with zero. Therefore, the first element is at position zero.

The firstElement() method returns the element at position zero but does *not* change the current element position.

## Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");

// This call returns the first element containing "This"
String anElement = MyString.firstElement();
```

## See also

```
lastElement()
```

---

## getElementCount()

Returns the total number of elements in the DtpSplitString object.

## Syntax

```
int getElementCount()
```

## Return values

Returns an integer containing the total number of elements.

## Notes

Elements are numbered from first to last beginning with zero. If getElementCount() returns 6, the highest-numbered element is 5.

## Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");

// This call returns the integer 4
String numElements = MyString.getElementCount();
```

## See also

```
firstElement(), lastElement()
```

---

## getEnumeration()

Returns an Enumeration of String objects where each String is one of the parsed tokens.

### Syntax

```
Enumeration getEnumeration()
```

### Return values

Returns an Enumeration object.

### Notes

The `getEnumeration()` method provides another way to process the parsed tokens in a `DtpSplitString` object. For more information on working with Enumeration objects, see the `Java.Util` package.

---

## lastElement()

Returns the last element in the `DtpSplitString` object.

### Syntax

```
String lastElement()
```

### Return values

Returns a String containing the last element.

### Exceptions

`DtpNoElementAtPositionException` – If there are no elements.

### Notes

Elements are numbered from first to last beginning with zero. The last element is the highest-numbered element. The position number of the last element is equivalent to `getElementCount()-1`.

The `lastElement()` method returns the last element but does not change the current element position.

### Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This, is a test", ", ");

// This call returns the last element, containing "test"
String anElement = MyString.lastElement();
```

### See also

`firstElement()`, `getElementCount()`

---

## nextElement()

Returns the next element in the `DtpSplitString` object.



## Syntax

```
String nextElement()
```

## Return values

Returns a String containing the next element.

## Exceptions

`DtpNoElementException` – If there is no next element.

## Notes

The first time you call `nextElement()`, it returns the element at position zero. In subsequent method calls, `nextElement()` returns the element at position one, two, three, and so on. You can use `nextElement()`, along with `prevElement()`, to navigate the elements (tokens) in a `DtpSplitString` object.

## Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");

// This call returns element 0 containing "This"
String firstElement = MyString.nextElement()

// This call returns element 1 containing "is"
String secondElement = MyString.nextElement()
```

## See also

```
prevElement(), reset()
```

---

## prevElement()

Returns the previous element in the `DtpSplitString` object.

## Syntax

```
String prevElement()
```

## Return values

Returns a String containing the previous element.

## Exceptions

`DtpNoElementException` – If there is no previous element.

## Notes

You can use `prevElement()`, along with `nextElement()`, to navigate the elements (tokens) in a `DtpSplitString` object. The first time you call `nextElement()`, the element position is zero. Subsequent calls to `nextElement()` increment the position by one. The `prevElement()` method returns the previous element and decrements the element position by one.

## Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");

// This call returns element 0 containing "This"
String firstElement = MyString.nextElement()

// This call returns element 1 containing "is"
String secondElement = MyString.nextElement()

// This call returns element 0 containing "This"
String anotherElement = MyString.prevElement()
```

## See also

`nextElement()`

---

## reset()

Resets the current position number in the `DtpSplitString` object to zero.

## Syntax

```
void reset()
```

## Return values

None.

## Notes

The default element position is zero. Each time you call `nextElement()`, the element position increments by one. The `prevElement()` method returns the previous element and decrements the element position by one. You can use `reset()` to reset the current position back to zero.

## Examples

```
// Create a DtpSplitString object
DtpSplitString MyString = new DtpSplitString("This,is a test",", ");

// This call returns element 0 containing "This"
String firstElement = MyString.nextElement()

// This call returns element 1 containing "is"
String secondElement = MyString.nextElement()

// Reset the position to zero
MyString.reset()

// This call returns element 0 containing "This"
String firstElement = MyString.nextElement()
```

## See also

`nextElement()`, `prevElement()`

---

## Chapter 20. DtpUtils class

The DtpUtils class performs several general-purpose operations.

Table 138 summarizes the methods of the DtpUtils class.

Table 138. DtpUtils method summary

Method	Description	Page
padLeft()	Pads the string with the specified character.	433
padRight()	Pads the string with the specified character.	433
stringReplace()	Replaces all occurrences of a pattern within a string with another pattern.	433
truncate()	Truncates this number.	435

---

### padLeft()

Pads the string with the specified character.

#### Syntax

```
public static String padLeft(String src, char padWith, int totalLen)
```

#### Parameters

<i>src</i>	The string to be padded.
<i>padWith</i>	The character used in padding.
<i>totalLen</i>	The new size of the string, a positive number. If the value is 0, smaller than the size of the original string, or a negative number, the original string is returned.

#### Return values

A new padded string.

#### Notes

Pads the string with a specified character.

#### Examples

The following call returns 0000012345:

```
padLeft("12345", '0', 10);
```

The following call returns 123456:

```
padLeft("123456", '0', 5);
```

---

### padRight()

Pads the string with the specified character.

#### Syntax

```
public static String padLeft(String src, char padWith, int totalLen)
```

## Parameters

<i>src</i>	The string to be padded.
<i>padWith</i>	The character used in padding.
<i>totalLen</i>	The new size of the string, a positive number. If the value is 0, smaller than the size of the original string, or a negative number, the original string is returned.

## Return values

A new padded string.

## Notes

Pads the string with a specified character.

## Examples

The following call returns 1234500000:

```
padRight("12345", '0', 10);
```

The following call returns 123456:

```
padRight("123456", '0', 5);
```

---

## stringReplace()

Replaces all occurrences of a pattern within a string with another pattern.

## Syntax

```
public static String stringReplace(String src, String oldPattern,  
    String newPattern)
```

## Parameters

<i>src</i>	The string to change.
<i>oldPattern</i>	The character used in padding.
<i>newPattern</i>	The string pattern to use in replacement.

## Return values

A new string with the new pattern.

## Notes

The method replaces all occurrences of the value specified by *oldPattern* with the value specified by *newPattern*. For single character replacement, use the `replace()` in the Java `String` class. If *oldPattern* is not found, the original, unmodified string is returned.

## Examples

The following results in youyou and dad.

```
stringReplace("momomom and dad", "mom", "you");
```

---

## truncate()

Truncates this number.

### Syntax

```
public static double truncate(Object aNumber, int precision)
    throws DtpIncompatibleFormatException

public static double truncate(float aNumber, int precision)
public static double truncate(double aNumber, int precision)

public static int truncate(Object aNumber)
    throws DtpIncompatibleFormatException

public static int truncate(float aNumber)
public static int truncate(double aNumber)
```

### Parameters

*aNumber*            A number. The valid types are String, float, and double.  
*precision*        The number of digits to the right of the decimal to be removed.

### Return values

A double or int number.

### Notes

This method removes digits from this number, starting from the right.

The first three forms of the methods truncate the number by removing the digits to the right of the decimal place, starting from the right. If the input number is an integer, it will not get truncated. The number of type Object must be either String, Double or Float.

The last three forms of the methods truncate the number by removing all digits to the right of the decimal and return the int value.

### Examples

The following returns 123.45:

```
truncate("123.4567", 2);
```

The following returns 123:

```
truncate(123.456, 4)
```



---

## Chapter 21. IdentityRelationship class

The methods documented in this chapter operate on objects of the IdentityRelationship class. These objects represent instances of identity relationships. The IdentityRelationship class provides additional functionality needed when accessing the repository database. It combines a set of existing APIs into methods that provide ease of use for the map developer.

The source code for the methods in the IdentityRelationship class is provided and can be used as is in the IBM WebSphere InterChange Server environment, or can be customized to fit other environments.

Table 139 lists the methods of the IdentityRelationship class.

Table 139. IdentityRelationship method summary

Method	Description	Page
addMyChildren()	Adds the specified child instances to a parent/child relationship for an identity relationship.	437
deleteMyChildren()	Removes the specified child instances to a parent/child relationship for an identity relationship belonging to the specified parent.	439
foreignKeyLookup()	Performs a lookup in a foreign relationship table based on the foreign key of the source business object, failing to find a relationship instance if the foreign key does not exist in the foreign relationship table.	440
foreignKeyXref()	Performs a lookup in the relationship table in the relationship database based on the foreign key of the source business object, adding a new relationship instance in the foreign relationship table if the foreign key does not exist.	442
maintainChildVerb()	Sets the child business object verb based on the map execution context and the verb of the parent business object.	444
maintainCompositeRelationship()	Maintains a composite identity relationship from within the parent map.	446
maintainSimpleIdentityRelationship()	Maintains a simple identity relationship from within either a parent or child map.	448
updateMyChildren()	Adds and deletes child instances in a specified parent/child relationship of an identity relationship as necessary.	450

**Note:** All methods in the IdentityRelationship class are declared as static. You can call any of the methods in this class from an existing relationship instance or by referencing the IdentityRelationship class: IdentityRelationship.*method*, where *method* is the name of a method in Table 139.

---

### addMyChildren()

Adds the specified child instances to a parent/child relationship for an identity relationship.

## Syntax

```
public static void addMyChildren(String parentChildRelDefName,
                                String parentParticpntDefName, BusObj parentBusObj,
                                String childParticpntDefName, Object childBusObjList,
                                CxExecutionContext map_ctx)
```

## Parameters

*parentChildRelDefName*

The name of the parent/child relationship definition.

*parentParticpntDefName*

The name of the participant definition that represents the parent business object in the parent/child relationship.

*parentBusObj*

The variable that contains the parent business object.

*childParticpntDefName*

The name of the participant definition that represents the child business object in the parent/child relationship.

*childBusObjList*

The variable that contains child business object or objects to be added to the relationship. This parameter can be either a single generic business object (BusObj) or an array of generic business objects (BusObjArray).

*map\_ctx*

The map execution context. To pass the map execution context, use the `cxExecCtx` variable, which Map Designer defines for every map.

## Return values

None.

## Exceptions

RelationshipRuntimeException

## Notes

The `addMyChildren()` method adds the child instances in *childBusObjList* to the relationship tables of the *parentChildRelDefName* relationship definition. This method is useful in a custom relationship involving a parent business object with a unique key. When a parent business object has the addition of new child objects, use `addMyChildren()` to compare the after-image (in *parentBusObj*) with the before-image (information in the relationship tables) to determine which child objects in the after-image are new. For each new child object, `addMyChildren()` adds a child instance to the relationship tables for the parent and child participants (*parentParticpntDefName* and *childParticpntDefName*, respectively). If the parent business object does not exist in the relationship table, `addMyChildren()` inserts a relationship instance for this parent object.

The `addMyChildren()` method requires that a parent/child relationship be defined with Relationship Designer. For information on how to create this kind of relationship, see “Creating the parent/child relationship definition” on page 282.



## See also

`deleteMyChildren()`, `updateMyChildren()`

“Managing child instances” on page 282.

---

## deleteMyChildren()

Removes the specified child instances to a parent/child relationship for an identity relationship belonging to the specified parent.

### Syntax

```
void deleteMyChildren(String parentChildRelDefName,  
    String parentParticipntDefName, BusObj parentBusObj,  
    String childParticipntDefName, Object childBusObjList,  
    CxExecutionContext map_ctx)
```

```
void deleteMyChildren(String parentChildRefDefName,  
    String parentParticipntDefName, BusObj parentBusObj,  
    String childParticipntDefName, CxExecutionContext map_ctx)
```

### Parameters

*parentChildRelDefName*

The name of the parent/child relationship definition.

*parentParticipntDefName*

The name of the participant definition that represents the parent business object in the parent/child relationship.

*parentBusObj*

The variable that contains the parent business object.

*childParticipntDefName*

The name of the participant definition that represents the child business object in the parent/child relationship.

*childBusObjList*

The variable that contains child business object or objects to be deleted from the relationship. This parameter can be either a single generic business object (BusObj) or an array of generic business objects (BusObjArray).

*map\_ctx*

The map execution context. To pass the map execution context, use the `cxExecCtx` variable, which Map Designer defines for every map.

### Return values

None.

### Exceptions

`RelationshipRuntimeException`

### Notes

The `deleteMyChildren()` method deletes child instances from a parent/child *parentChildRelDefName* relationship definition. It supports the following forms:

- The first form of the method removes from the relationship tables for the parent and child participants those child instances that correspond to each of the child

business objects in *childBusObjList*. It locates a child instance to delete by matching the child object's value and name, as well as the parent object's value and name.

- The second form of the method removes from relationship tables for the parent and child participants all child instances for the *parentBusObj* parent object. It locates the child instance to delete by matching the parent object's value and name.

This method is useful in a custom relationship involving a parent business object with a unique key. When a parent business object has removed child objects, use `deleteMyChildren()` to compare the after-image (in *parentBusObj*) with the before-image (information in the relationship tables) to determine which child objects in the after-image have been removed. For each child object, `deleteMyChildren()` removes the corresponding child instance from the relationship tables for the parent and child participants (*parentParticpntDefName* and *childParticpntDefName*, respectively).

The `deleteMyChildren()` method requires that a parent/child relationship be defined with Relationship Designer. For information on how to create this kind of relationship, see "Creating the parent/child relationship definition" on page 282.

## See also

`addMyChildren()`, `updateMyChildren()`

"Managing child instances" on page 282

---

## foreignKeyLookup()

Performs a lookup in a foreign relationship table based on the foreign key of the source business object, failing to find a relationship instance if the foreign key does not exist in the foreign relationship table.

## Syntax

```
public static void foreignKeyLookup(String relDefName,
    String appParticpntDefName, BusObj
    appSpecificBusObj, String appForeignAttr,
    BusObj genericBusObj,
    String genForeignAttr, CwExecutionContext map_ctx)
```

## Parameters

*relDefName* The name of the simple identity relationship that manages the foreign business object.

*appParticpntDefName* The name of the participant definition that represents the application-specific business object in the simple identity relationship. The type of this participant is the foreign application-specific business object.

*appSpecificBusObj* The variable that contains the application-specific business object, which contains the reference to the foreign business object.

*appForeignAttr* The name of the attribute in the application-specific business object that contains a key value for the foreign business object.

<i>genericBusObj</i>	The variable that contains the generic business object to or from which the <i>appSpecificObject</i> is being mapped.
<i>genForeignAttr</i>	The name of the attribute name in the generic business object that contains the generic reference to a foreign business object.
<i>map_ctx</i>	The map execution context. To pass the map execution context, use the <i>cwExecCtx</i> variable, which Map Designer defines for every map.

## Return values

None.

## Exceptions

RelationshipRuntimeException

## Notes

The `foreignKeyLookup()` method performs a foreign key lookup on the relationship table for the *AppParticipantDefName* participant; that is, it checks the foreign relationship table for a relationship instance that matches the value in the foreign key of the *appSpecificBusObj* business object. If this lookup fails, the `foreignKeyLookup()` method just sets the foreign key in the destination business object to null; it does *not* insert a row in the foreign relationship table (as the `foreignKeyXref()` method does). This method can be used in both inbound and outbound maps.

## Examples

On the `Clarify_PartRequest` to `Requisition` object, the `VendorId` field is a foreign key lookup. This is because Purchasing does not call `Vendor Wrapper`. We do not use the `foreignKeyXref()` method here because we do not want to insert a row if the lookup fails.

```

if (ObjCustomerRole.isNull("RoleId"))
{
    logError(5003, "OrderAssociatedCustomers.RoleId");
    // throw new MapFailureException("OrderAssociatedCustomers.RoleId
    // is null");
}

try
{
    IdentityRelationship.foreignKeyLookup("Customer", "SAPCust",
        ObjSAP_OrderPartners, "PartnerId", ObjCustomerRole,
        "RoleId", cwExecCtx);
}

catch (RelationshipRuntimeException re)
{
    logWarning(re.getMessage());
}

if (ObjSAP_OrderPartners.get("PartnerId") == null)
{
    logError(5007, "SAP_OrderPartners.PartnerId",
        "OrderAssociatedCustomers.RoleId", "Customer", "SAPCust",
        strInitiator);
    throw new MapFailureException("ForeignKeyLookup failed");
}

```

## See also

`foreignKeyXref()`

“Performing foreign key lookups” on page 290

---

## foreignKeyXref()

Performs a lookup in the relationship table in the relationship database based on the foreign key of the source business object, adding a new relationship instance in the foreign relationship table if the foreign key does not exist.

## Syntax

```
public static void foreignKeyXref(String relDefName,  
    String appParticpntDefName, String genParticpntDefName,  
    BusObj appSpecificBusObj, String appForeignAttr,  
    BusObj genericBusObj, String genForeignAttr,  
    CxExecutionContext map_ctx)
```

## Parameters

*relDefName* The name of the simple identity relationship name that manages the foreign business object.

*appParticpntDefName* The name of the participant definition for the application-specific business object in the simple identity relationship. The type of this participant is the foreign application-specific business object.

*genParticpntDefName* The name of the participant definition for the generic business object in the simple identity relationship. The type of this participant is the foreign generic business object.

*appSpecificBusbj* The application-specific business object that contains the reference to the foreign object.

*appForeignAttr* The name of the attribute in the application-specific business object that contains a key value for the foreign business object.

*genericObject* The generic business object to or from which the *appSpecificObject* is being mapped.

*genForeignAttr* The name of the attribute name in the generic business object that contains the generic reference to a foreign business object.

*map\_ctx* The map execution context. To pass the map execution context, use the `cxExecCtx` variable, which Map Designer defines for every map.

## Return values

None.

## Exceptions

`RelationshipRuntimeException`

## Notes

The `foreignKeyXref()` method performs a foreign key lookup on the relationship table for the `AppParticpntDefName` participant; that is, it checks the foreign relationship table for a relationship instance that matches the value in the foreign key of the `appSpecificBusObj` business object. If this lookup fails, the `foreignKeyXref()` method adds a new relationship instance for the application-specific key to the foreign relationship table; it does *not* just set the foreign key in the destination business object to null (as the `foreignKeyLookup()` method does). This method can be used in both inbound and outbound maps.

The `foreignKeyXref()` method performs the following validations on arguments that are passed in:

- Validate the name of the `relDefName` relationship definition.
- Validate the name of the `particpntDefName` participant definition for the application-specific business object.
- Make sure that the `relDefName` relationship is an identity relationship. In addition, the participant definition in `relDefName` that represents the generic business object must be defined as IBM WebSphere InterChange Server-managed. For more information on how to specify these settings, see “Defining identity relationships” on page 244.

If any of these validations fails, `foreignKeyXref()` throws the `RelationshipRuntimeException` exception.

Once the arguments are validated, the action that `foreignKeyXref()` takes depends on the following information:

- The calling context—in the map execution context, passed in as part of the `map_ctx` argument (`cwExecCtx`)
- The verb—in the source business object
  - Application-specific business object (`appSpecificBusObj`) for calling contexts `EVENT_DELIVERY` (or `ACCESS_REQUEST`) and `SERVICE_CALL_RESPONSE`
  - Generic business object (`genericBusObj`) for calling contexts `SERVICE_CALL_REQUEST` and `ACCESS_RESPONSE`

The `foreignKeyXref()` method handles all of the basic adding of relationship instances in the foreign relationship table for the appropriate combination of calling context and verb. For more information on the actions that `foreignKeyXref()` takes, see “Using the Foreign Key Cross-Reference function block” on page 291. Table 110 and Table 111 provide the actions for each of the calling contexts.

## Examples

On the `Clarify_SFAQuote to Order` map, the `CustomerId` field is a foreign key lookup. This is because `Sales Order Processing Collab` calls `Customer Wrapper`.

```
if (ObjSAP_OrderLineItem.get("SAP_OrderLineObjectIdentifier[0]")
    != null)
{
    if (ObjSAP_OrderLineItem.getString(
        "SAP_OrderLineObjectIdentifier[0].ObjectQualifier").equals("002"))
    {
        BusObj temp = ObjSAP_OrderLineItem.getBusObj(
            "SAP_OrderLineObjectIdentifier[0]");
        if (temp.isNull("ItemId"))
        {
            logWarning(5003,
```

```

    "SAP_OrderLineItem.SAP_OrderLineObjectIdentifier[1].ItemId");
    }
    else
    {
        try
        {
            IdentityRelationship.foreignKeyXref(
                "Item",
                "SAPMbasic",
                "CWIItba",
                temp,
                "ItemId",
                ObjOrderLineItem,
                "ItemId",
                cwExecCtx);
        }
        catch (RelationshipRuntimeException re)
        {
            logWarning(re.getMessage());
        }

        if (ObjOrderLineItem.get("ItemId") == null )
        {
            logError(5009, "OrderLineItem.ItemId",
                "SAP_OrderLineItem.SAP_OrderLineObjectIdentifier.ItemId",
                "Item",
                "SAPMbasic",
                strInitiator);

            throw new MapFailureException("ForeignKeyXref() failed");
        }
    }
}
}

```

## See also

`foreignKeyLookup()`

“Performing foreign key lookups” on page 290

---

## maintainChildVerb()

Sets the child business object verb based on the map execution context and the verb of the parent business object.

## Syntax

```

public static void maintainChildVerb (String relDefName,
    String appSpecificParticpntName,
    String genericParticpntName,
    BusObj appSpecificObj,
    String appSpecificChildObj,
    BusObj genericObj,
    String genericChildObj,
    CxExecutionContext map_ctx,
    boolean to_Retrieve,
    boolean is_Composite)

```

## Parameters

*relDefName*      The name of the identity relationship name that manages the child business object.

<i>appSpecificParticpntName</i>	The name of the application-specific participant definition.
<i>genericParticpntName</i>	The name of the generic participant definition.
<i>appSpecificObj</i>	The application-specific object that contains the child object.
<i>appSpecificChildObj</i>	The name of the application child business object.
<i>genericObj</i>	The generic business object to or from which the <i>appSpecificObject</i> is being mapped.
<i>genericChildObj</i>	The name of the generic child business object.
<i>ctx</i>	The execution context.
<i>to_Retrieve</i>	The flag for the SERVICE_CALL_RESPONSE logic. When the condition is true, update the verbs of the child business objects. If false, do nothing.
<i>isComposite</i>	The flag that indicates whether the child participant uses composite keys. If the condition is true, keys are used; if false, keys are not used.

## Return values

None.

## Exceptions

RelationshipRuntimeException—see the Notes section for more information on when this exception is thrown

ClassCastException

## Notes

The `maintainChildVerb()` method performs the following validations on arguments that are passed in:

- Validate the name of the *relDefName* relationship definition.
- Validate the name of the participant definitions for the application-specific business object (*appSpecificParticpntName*) and the generic business object (*genericParticpntName*).
- Make sure that the application-specific (*appSpecificObject*) and generic business objects (*genericObject*) are *not* null.
- Make sure that the *relDefName* relationship is an identity relationship. In addition, the participant definition in *relDefName* that represents the generic business object must be defined as IBM WebSphere InterChange Server-managed. For more information on how to specify these settings, see “Defining identity relationships” on page 244.

If any of these validations fails, `maintainChildVerb()` throws the `RelationshipRuntimeException` exception.

Once the arguments are validated, the action that `maintainChildVerb()` takes depends on the following information:

- The calling context—in the map execution context, passed in as part of the *map\_ctx* argument (*cxExecCtx*)
- The verb—in the source business object
  - Application-specific business object (*appSpecificObj*) for calling contexts `EVENT_DELIVERY` (or `ACCESS_REQUEST`) and `SERVICE_CALL_RESPONSE`
  - Generic business object (*genericObj*) for calling context `SERVICE_CALL_REQUEST`

For more information on the actions that `maintainChildVerb()` takes, see “Determining the child verb setting” on page 288. Table 105 through Table 108 provide the actions for each of the calling contexts.

You call this method in the transformation step for the child attribute of a parent object. This child object can participant in either

- In the transformation step for the key attribute of a submap that transforms child business objects if the child business objects are related using a unique key.

You usually use `maintainChildVerb()` to set the verb of a child object that participates in a composite identity relationship (`maintainCompositeRelationship()`). However, you can also call it to set the verb of a child object that participates in a simple identity relationship (`maintainSimpleIdentityRelationship()`).

## Examples

For an example involving `maintainChildVerb()`, see “Customizing map rules for a composite identity relationship” on page 277.

## See also

`maintainCompositeRelationship()`, `maintainSimpleIdentityRelationship()`

“Setting the source child verb” on page 287

---

## maintainCompositeRelationship()

Maintains a composite identity relationship from within the parent map.

### Syntax

```
public static void maintainCompositeRelationship(String relDefName,
String particpntDefName, BusObj appSpecificBusObj,
Object genericBusObjList, CxExecutionContext map_ctx)
```

### Parameters

*relDefName* The name of the composite identity relationship (as defined in Relationship Designer) in which the parent attribute participates.

*particpntDefName* The name of the participant that includes the composite key. This participant is always application-specific.

*appSpecificBusObj* The variable that contains the application-specific business object used in this map. This business object is the parent business object.

*genericBusObjList* The variable that contains the generic business object or objects



used in this map, each generic business object is a contained child business object of the generic parent object. This parameter can be either a single generic business object (*BusObj*) or an array of generic business objects (*BusObjArray*).

*map\_ctx*

The map execution context. To pass the map execution context, use the *cwExecCtx* variable, which Map Designer defines for every map.

## Return values

None.

## Exceptions

*RelationshipRuntimeException*

*CxMissingIDException*

If a participant does not exist in the relationship tables during a map execution with a verb of Retrieve and an calling context of *SERVICE\_CALL\_REQUEST*. The connector sends a “service call request failed” message to the collaboration without sending the business object to the application.

## Notes

The *maintainCompositeRelationship()* method maintains the relationship table associated with the *particpntDefName* participant of the *relDefName* composite identity relationship. This method maintains a relationship whose participant uses keys from multiple business objects at different levels (a composite key).

**Note:** The *maintainCompositeRelationship()* method *cannot* handle the case where the child’s composite key depends on its grandparents. For more information, see “Actions of General/APIs/Identity Relationship/Maintain Composite Relationship” on page 275.

This method iterates through all the child business objects in the *appSpecificObj* parent business object, maintaining the relationship instances in the *partDefName* participant’s relationship table. The method obtains the relationship instance IDs from the array of generic business objects that it receives (*genericObjs*). For each child instance, *maintainCompositeRelationship()* calls the *maintainSimpleIdentityRelationship()* method to perform the actual relationship-table management. The action that *maintainSimpleIdentityRelationship()* takes depends on the following information:

- The calling context—in the map execution context, passed in as part of the *map\_ctx* argument (*cwExecCtx*)
- The verb—in the source business object, which is either:
  - Application-specific business object (*appSpecificBusObj*) for calling contexts *EVENT\_DELIVERY* (or *ACCESS\_REQUEST*) and *SERVICE\_CALL\_RESPONSE*
  - Generic business object (one element of the *genericBusObjList* array) for calling contexts *SERVICE\_CALL\_REQUEST* and *ACCESS\_RESPONSE*

For more information on the actions that *maintainSimpleIdentityRelationship()* takes, see “Accessing identity relationship tables” on page 263. Table 95 through Table 99 provide the actions for each of the calling contexts.

Use `maintainCompositeRelationship()` in conjunction with the `maintainChildVerb()` and `updateMyChildren()` methods to maintain a composite relationship. For more information, see “Customizing map rules for a composite identity relationship” on page 277.

## Examples

```
// This is an example of a code fragment in a parent map. It maintains
// the relationship table for all instances of a child object type for
// this application-specific parent object.
```

```
BusObjArray secondLevel2 =
    (BusObjArray)ObjFirstLevelBusObj2.get("MultiCardChild");
```

```
IdentityRelationship.maintainCompositeRelationship(
    "CmpoSRel",
    "AppSpPrt",
    ObjFirstLevelBusObj2,
    secondLevel2,
    cwExecCtx);
```

```
IdentityRelationship.updateMyChildren(
    "PCRel",
    "Parent",
    ObjFirstLevelBusObj2,
    "Child",
    "MultiCardChild",
    "CmpoSRel",
    "AppSpPrt",
    cwExecCtx);
```

For more examples involving `maintainCompositeRelationship()`, see “Customizing map rules for a composite identity relationship” on page 277.

## See also

`updateMyChildren()`, `maintainChildVerb()`, `maintainSimpleIdentityRelationship()`

“Using composite identity relationships” on page 274

---

## maintainSimpleIdentityRelationship()

Maintains a simple identity relationship from within either a parent or child map.

### Syntax

```
public static void maintainSimpleIdentityRelationship(
    String relDefName, String particpntDefName,
    BusObj appSpecificBusObj, BusObj genericBusObj,
    CxExecutionContext map_ctx)
```

### Parameters

*relDefName*      The name of the simple identity relationship (as defined in Relationship Designer) in which this attribute participates.

*particpntDefName*      The name of the participant definition that represents the application-specific business object.

*appSpecificBusObj*      The variable that contains the application-specific business object used in this map.

<i>genericBusObj</i>	The variable that contains the generic business object used in this map.
<i>map_ctx</i>	The map execution context. To pass the map execution context, use the <i>cwExecCtx</i> variable, which Map Designer defines for every map.

## Return values

None.

## Exceptions

### RelationshipRuntimeException

see the Notes section for more information on when this exception is thrown.

### CxMissingIDException

If a participant does not exist in the relationship tables during a map execution with a verb of Retrieve and an calling context of `SERVICE_CALL_REQUEST`. The connector sends a “service call request failed” message to the collaboration without sending the business object to the application.

## Notes

The `maintainSimpleIdentityRelationship()` method maintains the relationship table associated with the *particpntDefName* participant of the *relDefName* simple identity relationship. This method maintains a relationship whose participant uses unique keys from multiple business objects at the same level.

The `maintainSimpleIdentityRelationship()` method performs the following validations on arguments that are passed in:

- Validate the name of the *relDefName* relationship definition.
- Validate the name of the *particpntDefName* participant definition for the application-specific business object.
- Make sure that the application-specific (*appSpecificBusObj*) and generic business objects (*genericBusObj*) are *not* null.
- Make sure that the *relDefName* relationship is an identity relationship. In addition, the participant definition in *relDefName* that represents the generic business object must be defined as IBM WebSphere InterChange Server-managed. For more information on how to specify these settings, see “Defining identity relationships” on page 244.
- Make sure the calling context is valid (see Table 94 for a list of valid calling contexts).
- Make sure that the application-specific business object’s verb is supported. It must be one of the following: Create, Update, Delete, Retrieve.

If any of these validations fails, `maintainSimpleIdentityRelationship()` throws the `RelationshipRuntimeException` exception.

Once the arguments are validated, the action that `maintainSimpleIdentityRelationship()` takes depends on the following information:

- The calling context—in the map execution context, passed in as part of the *map\_ctx* argument (*cwExecCtx*)

- The verb—in the source business object
  - Application-specific business object (*appSpecificBusObj*) for calling contexts EVENT\_DELIVERY (or ACCESS\_REQUEST) and SERVICE\_CALL\_RESPONSE
  - Generic business object (*genericBusObj*) for calling contexts SERVICE\_CALL\_REQUEST and ACCESS\_RESPONSE

The `maintainSimpleIdentityRelationship()` method handles all of the basic adding and deleting of participants and relationship instances for each combination of calling context and verb. For more information on the actions that `maintainSimpleIdentityRelationship()` takes, see “Accessing identity relationship tables” on page 263. Table 95 through Table 99 provide the actions for each of the calling contexts.

You can call this method in either of the following cases:

- In the transformation step for the key attribute of a parent object
- In the transformation step for the key attribute of a submap that transforms child business objects if the child business objects are related using a unique key.

Use `maintainSimpleIdentityRelationship()` in conjunction with the `maintainChildVerb()` method to maintain a simple identity relationship. For more information, see “Defining transformation rules for a simple identity relationship” on page 273.

## Examples

The following example maintains the simple identity relationship between the `Clarify_BusOrg` and generic `Customer` business objects in an inbound `Clarify_BusOrg-to-Customer` map:

```
IdentityRelationship.maintainSimpleIdentityRelationship(
    "CustIdentity",
    "ClarBusOrg",
    ObjClarify_BusOrg,
    ObjCustomer,
    cxExecCtx);
```

For more examples involving `maintainSimpleIdentityRelationship()`, see “Defining transformation rules for a simple identity relationship” on page 273.

## See also

`maintainChildVerb()`

“Using simple identity relationships” on page 263

---

## updateMyChildren()

Adds and deletes child instances in a specified parent/child relationship of an identity relationship as necessary.

### Syntax

```
void updateMyChildren(String parentChildRelDefName,
    String parentParticpntDef, BusObj parentBusObj,
    String childParticpntDef, String childAttrName,
    String childIdentityRelDefName,
    String childIdentityParticpntDefName,
    CxExecutionContext map_ctx)
```

## Parameters

*parentChildRelDefName*

The name of the parent/child relationship definition.

*parentParticipntDefName*

The name of the participant definition that represents the parent business object in the parent/child relationship.

*parentBusObj* The variable that contains the parent business object.

*childParticipntDefName*

The name of the participant definition that represents the child business object in the parent/child relationship.

*childAttrName* The name of the attribute in the parent business object whose type is the child object name that participates in the parent/child relationship. For example, in a customer-address relationship, if the parent object contains an Address1 attribute, which is a child business object of type Address, the *childAttrName* attribute name is Address1.

*childIdentityRelDefName*

The name of the identity relationship in which the child business object participates.

*childIdentityParticipntDefName*

The name of the participant definition that represents the child business object in the identity relationship.

*map\_ctx*

The map execution context. To pass the map execution context, use the *cwExecCtx* variable, which Map Designer defines for every map.

## Return values

None.

## Exceptions

RelationshipRuntimeException

see the Notes section for more information on when this exception is thrown

## Notes

The `updateMyChildren()` method updates the child instances in the relationship tables of the *parentChildRelDefName* and *childIdentityRelDefName* relationship definitions. This method is useful in an identity relationship when a parent business object has been updated as a result of the addition or removal of child objects. Use `updateMyChildren()` to compare the after-image (in *parentBusObj*) with the before-image (information in the relationship tables) to determine which child objects in the after-image are new or deleted.

**Note:** The `updateMyChildren()` method *cannot* handle the case where the child's composite key depends on its grandparents. For more information, see "Tips on using Update My Children" on page 284.

The `updateMyChildren()` method performs the following validations on arguments that are passed in:

- Validate the name of the *parentChildRelDefName* relationship definition (first argument).
- Make sure that the *parentChildRelDefName* relationship is a parent/child relationship and that the *parentParticpntDefName* and *childParticpntDefName* are part of the *parentChildRefDefName* relationship definition.
- Make sure that the *childIdentityRelDefName* relationship is an identity relationship. In addition, the participant definition in *childIdentityRelDefName* that represents the generic business object must be defined as IBM WebSphere InterChange Server- managed. For more information on how to specify these settings, see “Defining identity relationships” on page 244.
- Make sure that the *childIdentityParticpntDefName* is part of the *childIdentityRefDefName* relationship definition

If any of these validations fails, `updateMyChildren()` throws the `RelationshipRuntimeException` exception.

Once the arguments are validated, the `updateMyChildren()` method adds children or deletes children from the list of child business objects that belong to the specified parent business object as appropriate. This method performs one of the following tasks to the relationship tables for the parent and child participants (*parentParticpntDefName* and *childParticpntDefName*, respectively):

- For each new child object, `updateMyChildren()` adds a child instance. This method does *not* add to the child’s relationship table because all the business objects that are currently associated with the parent object have already been maintained when `maintainCompositeRelationship()` was called.
- For each deleted child object, `updateMyChildren()` removes the corresponding child instance. This method removes from the child’s cross-reference table in addition to the parent/child relationship table.

The `updateMyChildren()` method requires that a parent/child relationship is defined with Relationship Designer. For information on how to create this kind of relationship, see “Creating the parent/child relationship definition” on page 282.

**Note:** If the child business object has a unique key, the child participant’s attribute is the unique key of the child object. If the child object does not have a unique key, the child participant’s attribute is this nonunique key.

## Examples

For an example involving `updateMyChildren()` in conjunction with the `maintainCompositeRelationship()` method, see the Examples section of `maintainCompositeRelationship()`.

For more examples involving `updateMyChildren()`, see “Customizing map rules for a composite identity relationship” on page 277.

## See also

`addMyChildren()`, `deleteMyChildren()`, `maintainCompositeRelationship()`, `maintainSimpleIdentityRelationship()`

“Handling updates to the parent business object” on page 283

---

## Chapter 22. CxExecutionContext class

The CxExecutionContext class provides methods for operating on the global execution context, which is a holder for user-accessible context information that is associated with a given flow. Currently, only the map-specific execution information is shown as the map execution context. Map Designer automatically declares a special variable in the map code to access the map execution context, `cxExecCtx`. When you call a map from within a collaboration, however, you must instantiate your own global execution context and map execution context.

Table 140 summarizes the methods of the CxExecutionContext class.

Table 140. CxExecutionContext method summary

Method	Description	Page
CxExecutionContext()	Constructs a new instance of a global execution context.	453
getContext()	Retrieves the specified execution context from the global execution context.	454
setContext()	Sets a particular execution context to be part of the global execution context.	454

---

### Static constants

The CxExecutionContext class defines the static constants that Table 141 shows.

Table 141. Static constants defined in the CxExecutionContext class

Constant name	Meaning
MAPCONTEXT	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.

---

## getContext()

Retrieve the specified execution context from the global execution context.

### Syntax

```
Object getContext(String contextName)
```

### Parameters

*contextName* The name of an 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);
```

---

## 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

### Notes

You might explicitly set the map execution context if you want to execute a submap with a relationship transformation. In this case, you would choose to create a new context with its own calling context (initiator) in the map context so that the relationship behaves correctly.

Currently, you can only use the CxExecutionContext() to set the map execution context, and only save one MapExeContext() instance in the CxExecutionContext(). If you use setContext() to set a new MapExeContext() into the CxExecutionContext(), the old one will be gone.

### Examples

The following example shows the use of setContext():



```

        mapExeContext originalMapExeContext = (MapExeContext)
cwExecCtx.getContext(CxExecutionContext.MAPCONTEXT);
        MapExeContext newMapExeContext = new MapExeContext();
newMapExeContext.setInitiator(originalMapExeContext.getInitiator());
        newMapExeContext.setConnName(originalMapExeContext.getConnName());
newMapExeContext.setLocale(originalMapExeContext.getLocale());
        cwExecCtx.setContext(CxExecutionContext.MAPCONTEXT,
newMapExeContext);

```

The following example saves a map execution context into a global execution context:

```

globalExeContext.setContext(CxExecutionContext.MAPCONTEXT,
        mapExeContext);

```

The following example shows how to set the map execution context:

```

CxExecutionContext cwCtx = new CxExecutionContext();
MapExeContext mapCtx = new MapExeContext();
cwCtx.setContext(CxExecutionContext.MAPCONTEXT, mapCtx);

// do some work involving execution context
        cwExecCtx.setContext(CxExecutionContext.MAPCONTEXT,
originalMapExeContext);

```



---

## Chapter 23. MapExeContext class

The MapExeContext class provides methods for querying and setting various runtime values that are in effect during map execution.

Table 142 summarizes the methods of the MapExeContext class.

Table 142. MapExeContext method summary

Method	Description	Page
getConnName()	Retrieves the connector name associated with the current map instance.	457
getInitiator()	Retrieves the calling context associated with the current map instance.	457
getLocale()	Retrieves the locale associated with the map execution context.	458
getOriginalRequestBO()	Retrieves the original-request business object associated with the current map instance.	459
setConnName()	Sets the connector name associated with the current map instance.	460
setInitiator()	Sets the calling context associated with the current map instance.	460
setLocale()	Sets the locale associated with the map execution context.	460

---

### getConnName()

Retrieves the connector name associated with the current map instance.

#### Syntax

```
String getConnName()
```

#### Parameters

None.

#### Return values

Returns a String containing the connector name.

#### Exceptions

None.

#### See also

```
setConnName()
```

---

### getInitiator()

Retrieves the calling context associated with the current map instance.

#### Syntax

```
String getInitiator()
```

## Parameters

None.

## Return values

Returns a static constant variable representing the calling context for the execution of the current map instance. Calling contexts are one of the following values:

EVENT\_DELIVERY

The source business objects being mapped are sent from an application to InterChange Server through a connector.

ACCESS\_REQUEST

The source objects being mapped are sent from an application to InterChange Server through an access client.

SERVICE\_CALL\_REQUEST

The source objects being mapped are sent from InterChange Server to an application through a connector.

SERVICE\_CALL\_RESPONSE

The source objects being mapped are sent back to InterChange Server from an application through a connector after a successful service call request.

SERVICE\_CALL\_FAILURE

The source objects being mapped are sent back to InterChange Server from an application through a connector after a failed service call request.

ACCESS\_RESPONSE

The source objects being mapped are sent back from InterChange Server to the application through an access client.

## Exceptions

None.

## Notes

The calling context is part of the map execution context. For more information on how calling contexts are used in maps, see "Understanding map execution contexts" on page 189.

## Examples

In the following example, compare the map run-time initiator with the constants defined in the MapExeContext class:

```
cwMapCtx =  
(MapExeContext)cwExecCtx.getContext (CxExecutionContext.MAPCONTEXT);  
String sInitiator = null;  
sInitiator = cwMapCtx.getInitiator();  
logInfo("*****Initiator = + sInitiator);
```

## See also

getOriginalRequestBO(), setInitiator()

---

## getLocale()

Retrieves the locale associated with the map execution context.

## Syntax

```
Locale getLocale()
```

## Parameters

None.

## Return values

Returns a Locale object that contains the language and country code for the map execution context.

## Exceptions

None.

## Notes

This method must be run on the map variable of MapExeContext type, which is named cwMapCtx when generated by the system, or which you name when calling a map in an environment that does not automatically generate map code (such as within a collaboration).

## Examples

The following example retrieves the locale of the map execution context into a variable and then reports it with a trace statement:

```
Locale mapLocale = cwMapCtx.getLocale();
String mapLocaleToString = mapLocale.toString();
trace(3, "THE MAP LOCALE IS: " + mapLocaleToString);
```

## See also

```
setLocale()
```

---

## getOriginalRequestBO()

Retrieves the original-request business object associated with the current map instance.

## Syntax

```
BusObj getOriginalRequestBO()
```

## Parameters

None.

## Return values

Returns the original-request business object for the map, as the following table shows:

Calling Contexts	Original-Request Business Object
EVENT_DELIVERY, ACCESS_REQUEST	Application-specific business object that came in from the application
SERVICE_CALL_REQUEST, SERVICE_CALL_FAILURE	Generic business object that was sent down from InterChange Server
SERVICE_CALL_RESPONSE	Generic business object that was sent down by the SERVICE_CALL_REQUEST

---

Calling Contexts	Original-Request Business Object
ACCESS_RESPONSE	Application-specific business object that came in from the access request initially

---

## Exceptions

None.

## Notes

The original-request business object is part of the map execution context. The `getOriginalRequestBO()` method returns the original-request business object, which depends on the map's calling context. For more information on how this business object is used in maps, see "Original-request business objects" on page 192.

## See also

`getInitiator()`

---

## setConnName()

Sets the connector name associated with the current map instance.

## Syntax

```
void setConnName(String connectorName)
```

## Parameters

*connectorName* Name of the connector

## Return values

None.

## Exceptions

None.

## Notes

The controller for the connector you specify must be running in InterChange Server.

## See also

`getConnName()`

---

## setInitiator()

Sets the calling context associated with the current map instance.

## Syntax

```
void setInitiator(String callingContext)
```

## Parameters

*callingContext*

String containing one of the following values:

EVENT_DELIVERY	The source objects being mapped are sent from an application through a connector to InterChange Server.
ACCESS_REQUEST	The source objects being mapped are sent from an application to InterChange Server through an access client.
SERVICE_CALL_REQUEST	The source objects being mapped are sent from InterChange Server to an application through a connector.
SERVICE_CALL_RESPONSE	The source objects being mapped are sent back to InterChange Server from an application through a connector after a successful service call request.
SERVICE_CALL_FAILURE	The source objects being mapped are sent back to InterChange Server from an application through a connector after a failed service call request.
ACCESS_RESPONSE	The source objects being mapped are sent back from InterChange Server to the application through an access client.

## Return values

None.

## Exceptions

None.

## Notes

The calling context is part of the map execution context. The calling context indicates the direction in which the source business object is being mapped. For more information on how calling contexts are used in maps, see “Understanding map execution contexts” on page 189.

## See also

`getInitiator()`

---

## **setLocale()**

Sets the locale associated with the map execution context.

## Syntax

```
void setLocale(Locale newLocale)
```

## Parameters

*newLocale* The new Locale object to set the map execution context to.

## Return values

None.

## Exceptions

None.

## Notes

This method must be run on the map variable of MapExeContext type, which is named cwMapCtx when generated by the system, or which you name when calling a map in an environment that does not automatically generate map code (such as within a collaboration).

The locale of the business object produced by a map is affected by the local of the map's execution context. If you change the locale of the map execution context as part of the map's logic, therefore, the new locale is copied to the business object. This is done when the user-modifiable logic is finished executing (that is, when the transformations visible in the diagram of the Map Designer are finished). You can use this API to change the business object to a different locale than the one it had when it entered the map.

## Examples

The code below defines a new Locale object, sets the map execution context to that new Locale value, and then reports the map execution context locale:

```
Locale newLocale = new Locale("ja", "JP");  
cwMapCtx.setLocale(newLocale);  
trace(3, "THE MAP LOCALE IS NOW: " + cwMapCtx.getLocale().toString());
```

## See also

getLocale()

---

## Deprecated methods

Some methods in the MapExeContext class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but CrossWorlds recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 143 lists the deprecated method for the MapExeContext class. If you have not used Map Designer before, ignore this section.

Table 143. *Deprecated Method, MapExeContext Class*

Former method	Replacement
getGenericB0()	getOriginalRequestB0()



---

## Chapter 24. Participant class

The methods documented in this chapter operate on objects of the Participant class. Participant instances are used in relationship instances. Each Participant instance contains the following information:

- name of the relationship definition
- relationship instance ID
- name of the participant definition
- data to associate with the participant

The Participant class provides methods for setting and retrieving each of these values for a given participant.

Table 144 summarizes the methods of the Participant class.

*Table 144. Participant method summary*

Method	Description	Page
Participant()	Creates a new Participant instance.	463
getBusObj(), getString(), getLong(), getInt(), getDouble(), getFloat(), getBoolean()	Retrieves the data associated with the Participant instance.	465
getInstanceId()	Retrieves the relationship instance ID of the relationship in which the Participant instance is participating.	465
getParticipantDefinition()	Retrieves the participant definition name from which the Participant instance is created.	466
getRelationshipDefinition()	Retrieves the name of the relationship definition in which the Participant instance is participating.	466
set()	Sets the data associated with the Participant instance.	467
setInstanceId()	Sets the instance ID of the relationship in which the Participant instance is participating.	467
setParticipantDefinition()	Sets the participant definition name from which the Participant instance is created.	468
setRelationshipDefinition()	Sets the relationship definition in which the Participant instance is participating.	468

---

### Participant()

Creates a new Participant instance.

## Syntax

To add a new participant instance to an existing participant in a relationship instance:

```
Participant(String relDefName, String partDefName,
int instanceId, BusObj partData)
Participant(String relDefName, String partDefName,
int instanceId, String partData)
Participant(String relDefName, String partDefName,
int instanceId, long partData)
Participant(String relDefName, String partDefName,
int instanceId, int partData)
Participant(String relDefName, String partDefName,
int instanceId, double partData)
Participant(String relDefName, String partDefName,
int instanceId, float partData)
Participant(String relDefName, String partDefName,
int instanceId, boolean partData)
```

To create a new participant instance with no relationship instance:

```
Participant(String relDefName, String partDefName, BusObj partData)
Participant(String relDefName, String partDefName, String partData)
Participant(String relDefName, String partDefName, long partData)
Participant(String relDefName, String partDefName, int partData)
Participant(String relDefName, String partDefName, double partData)
Participant(String relDefName, String partDefName, float partData)
Participant(String relDefName, String partDefName, boolean partData)
```

## Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition that describes the participant.
<i>instanceId</i>	The relationship instance ID for the relationship instance to receive the new participant instance.
<i>participantData</i>	Data to associate with the participant instance. Can be one of the following data types: BusObj, String, long, int, double, float, boolean.

## Return values

Returns new participant instance.

## Exceptions

RelationshipRuntimeException – See “Handling exceptions” on page 186.

## Notes

This method is the Participant class constructor. It takes the following forms:

- The first form of the constructor adds a new participant instance to the relationship instance identified by *instanceId*.
- The second form creates a new participant instance with no associated relationship instance. You can use this participant instance as an argument to IdentityRelationship.addMyChildren() or Relationship.create() to create a new relationship instance. With the Relationship.create() method, having no relationship instance ID is a requirement.

The data to associate with the *participantData* parameter depends on the kind of relationship:

- To create a participant instance for an identity relationship, use a business object as the *participantData* parameter.
- To create a participant for a lookup relationship, use any of the following data types for the *participantData* parameter: String, long, int, double, float, boolean.

## Examples

```
// create a participant instance with no relationship instance ID
participant p = new Participant(myRelDef,myPartDef,myBusObj);

// create a relationship instance
int relInstanceId = Relationship.addParticipant(p);
```

## See also

`addMyChildren()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## `getBusObj()`, `getString()`, `getLong()`, `getInt()`, `getDouble()`, `getFloat()`, `getBoolean()`

Retrieves the data associated with the Participant instance.

## Syntax

```
BusObj getBusObj()
String getString()
long getLong()
int getInt()
double getDouble()
float getFloat()
boolean getBoolean()
```

## Return values

Returns the data associated with this participant instance. This data value is of the type included in the method name. For example, `getBoolean()` returns a boolean value, `getBusObj()` returns a `BusObj` value, `getDouble()` returns a double value, and so on.

## Exceptions

`RelationshipRuntimeException` – See “Handling exceptions” on page 186.

## See also

`set()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## `getInstanceld()`

Retrieves the relationship instance ID of the relationship in which the Participant instance is participating.

## Syntax

```
int getInstanceId()
```

## Return values

Returns an integer representing the instance ID of the relationship instance in which this Participant instance is participating. If the Participant instance is *not* a member of a relationship instance, this method returns the constant, `INVALID_INSTANCE_ID`.

## Exceptions

`RelationshipRuntimeException` – See “Handling exceptions” on page 186.

## See also

`setInstanceId()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## getParticipantDefinition()

Retrieves the participant definition name from which the Participant instance is created.

## Syntax

```
String getParticipantDefinition()
```

## Return values

Returns a `String` containing the name of the participant definition associated with this participant instance.

## Exceptions

`RelationshipRuntimeException` – See “Handling exceptions” on page 186.

## See also

`setParticipantDefinition()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## getRelationshipDefinition()

Retrieves the name of the relationship definition in which the Participant instance is participating.

## Syntax

```
String getRelationshipDefinition()
```

## Return values

Returns a `String` containing the name of the relationship definition in which this participant instance participates.

## Exceptions

`RelationshipRuntimeException` – See “Handling exceptions” on page 186.

## See also

`setRelationshipDefinition()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## set()

Sets the data associated with the Participant instance.

## Syntax

```
void set(BusObj partData)
void set(String partData)
void set(long partData)
void set(int partData)
void set(double partData)
void set(float partData)
void set(boolean partData)
```

## Parameters

*partData* Data to associate with the Participant instance. Can be one of the following data types: BusObj, String, long, int, double, float, boolean.

## Return values

None.

## Exceptions

RelationshipRuntimeException – See “Handling exceptions” on page 186.

## Notes

If you set the participant data to be a business object (BusObj type), the relationship definition and participant definition must already be set. If you set the participant data to any other data type, it does not matter which setting you specify first.

## See also

`getBusObj()`, `getString()`, `getLong()`, `getInt()`, `getDouble()`, `getFloat()`, `getBoolean()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## setInstanceId()

Sets the instance ID of the relationship in which the Participant instance is participating.

## Syntax

```
void setInstanceId(int id)
```

## Parameters

*id* Instance ID of the relationship.

## Return values

None.

## Exceptions

RelationshipRuntimeException – See “Handling exceptions” on page 186.

## Notes

One use of `setInstanceId()` is to remove the relationship instance ID when you want to pass a participant instance as a parameter to the `Participant()` or `create()` methods. In this case, you set the instance ID to the constant `INVALID_INSTANCE_ID`.

## Examples

```
// wipe out the relationship instance ID
myParticipant.setInstanceId(Participant.INVALID_INSTANCE_ID);

// pass the participant instance to the create() method
int newRelId = create(myParticipant);
```

## See also

`getInstanceId()`, Chapter 7, “Creating relationship definitions,” on page 237,  
“Transforming with a submap” on page 43

---

## setParticipantDefinition()

Sets the participant definition name from which the `Participant` instance is created.

## Syntax

```
void setParticipantDefinition(String partDefName)
```

## Parameters

*partDefName*      Name of the participant definition from which the `Participant` instance is created.

## Return values

None.

## Exceptions

RelationshipRuntimeException – See “Handling exceptions” on page 186.

## See also

`setParticipantDefinition()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## setRelationshipDefinition()

Sets the relationship definition in which the `Participant` instance is participating.

## Syntax

```
void setRelationshipDefinition(String relDefName)
```

## Parameters

*relDefName*      Name of the relationship definition.

## Return values

None.

## Exceptions

RelationshipRuntimeException – See “Handling exceptions” on page 186.

## See also

`getRelationshipDefinition()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43





---

## Chapter 25. Relationship class

The methods documented in this chapter operate on objects of the IBM WebSphere InterChange Server-defined class `Relationship`. The `Relationship` class provides methods for manipulating the runtime instances of relationships, called *relationship instances*. You typically use these methods in transformation steps for business object attributes that are mapped as identity relationships or static lookups. For more information on programming relationship attributes using the methods in this class, see “Transforming with a submap” on page 43.

Most methods in this class support variations in the parameters you specify. The variations generally follow these guidelines:

- To identify a specific participant in a relationship instance, you usually specify the relationship definition name, the participant definition name, the relationship instance ID, and the business object associated with the participant.
- Alternatively, you can specify a `Participant` instance which contains the relationship definition name, participant definition name, instance ID and business object, as its attributes.
- For some operations, you can omit the relationship instance ID (for example, when creating a new relationship) or the business object name.

In most cases, if you have a `Participant` instance (for example, as the result of a `retrieve()` call), it is easier to pass it as a parameter to a `Relationship` class method instead of specifying each attribute individually.

All methods in this class are declared as static. You can call them from existing relationship instances or by referencing the `Relationship` class.

Table 145 summarizes the methods in the `Relationship` class.

Table 145. Relationship method summary

Method	Description	Page
<b>Static methods</b>		
<code>addParticipant()</code>	Adds a new participant to a relationship instance.	472
<code>create()</code>	Creates a new relationship instance.	474
<code>deactivateParticipant()</code>	Deactivates a participant from one or more relationship instances.	475
<code>deactivateParticipantByInstance()</code>	Deactivates a participant from a specific relationship instance.	476
<code>deleteParticipant()</code>	Removes a participant instance from one or more relationship instances.	477
<code>deleteParticipantByInstance()</code>	Removes a participant from a specific relationship instance.	478
<code>getNewID()</code>	Returns the next available relationship instance ID for a relationship, based on the relationship definition name.	479
<code>retrieveInstances()</code>	Retrieves only the relationship instance IDs for zero or more relationship instances which contain a given participant instance.	480
<code>retrieveParticipants()</code>	Retrieves zero or more participants from a relationship instance.	482

Table 145. Relationship method summary (continued)

Method	Description	Page
updateParticipant()	Updates a participant in one or more relationship instances.	483
updateParticipantByInstance()	Updates a participant in a specific relationship instance.	484

## addParticipant()

Adds a new participant to a relationship instance.

### Syntax

To add a new participant to an existing relationship instance:

```
int addParticipant
  (String relDefName,
   String partDefName,
   int instanceId, BusObj partData)

int addParticipant
  (String relDefName,
   String partDefName,
   int instanceId, String partData)

int addParticipant
  (String relDefName,
   String partDefName, int instanceId,
   long partData)

int addParticipant
  (String relDefName,
   String partDefName, int instanceId,
   int partData)

int addParticipant
  (String relDefName,
   String partDefName,
   int instanceId,
   double partData)

int addParticipant
  (String relDefName,
   String partDefName,
   int instanceId, float partData)

int addParticipant
  (String relDefName,
   String partDefName,
   int instanceId,
   boolean partData)
```

To add a participant to a new relationship instance:

```
int addParticipant
  (String relDefName,
   String partDefName,
   BusObj partData)
int addParticipant
  (String relDefName,
   String partDefName,
   String partData)
int addParticipant
  (String relDefName,
```

```

String partDefName,
    long partData)
int addParticipant
    (String relDefName,
String partDefName,
    int partData)
int addParticipant
    (String relDefName,
String partDefName,
    double partData)
int addParticipant
    (String relDefName,
String partDefName,
    float partData)
int addParticipant
    (String relDefName,
String partDefName,
    boolean partData)

```

To add an existing participant instance to a relationship instance:

```
int addParticipant(Participant participant)
```

## Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition.
<i>instanceId</i>	Relationship instance ID of the relationship instance to receive the new participant.
<i>partData</i>	Data to associate with the participant. Can be one of the following data types: <code>BusObj</code> , <code>String</code> , <code>long</code> , <code>int</code> , <code>double</code> , <code>float</code> , <code>boolean</code> .
<i>participant</i>	Participant to add to the relationship.

## Return values

Returns an integer representing the instance ID of the relationship to receive the new participant.

## Exceptions

`RelationshipRuntimeException` – See “Handling exceptions” on page 186.

## Notes

The first form of the method adds a new participant to the relationship instance you specify. Each variation supports a different data type for the data to associate with the participant.

The second form, since it does not specify a relationship instance, creates a new relationship instance and adds the new participant. In this case, the return value is the instance ID of the newly created relationship instance. Each variation supports a different data type for the data to associate with the participant.

The third form adds the participant instance you pass to the relationship instance specified in the participant instance. If the participant instance has no relationship instance ID, a new relationship instance is created and the new instance ID is returned.

The `addParticipant()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the `Relationship` class.

## See also

`create()`

---

## create()

Creates a new relationship instance.

## Syntax

```
int create(String relDefName, String partDefName, BusObj partData)
int create(String relDefName, String partDefName, String partData)
int create(String relDefName, String partDefName, long partData)
int create(String relDefName, String partDefName, int partData)
int create(String relDefName, String partDefName, double partData)
int create(String relDefName, String partDefName, float partData)
int create(String relDefName, String partDefName, boolean partData)
int create(Participant participant)
```

## Parameters

*relDefName*      Name of the relationship definition.

*partDefName*     The name of the participant definition.

*partData*        Data to associate with the participant. Can be one of the following data types: `BusObj`, `String`, `long`, `int`, `double`, `float`, `boolean`.

*participant*     First participant in the relationship.

## Return values

Returns an integer representing the relationship instance ID of the new relationship.

## Exceptions

`RelationshipRuntimeException`

## Notes

The `create()` method creates a new relationship instance with one participant instance of the *partDefName* participant definition. You can specify the data for this new participant instance with the *partData* argument. After calling this method, you can call `addMyChildren()` to add more participants to the relationship instance.

In the last form of the method, the *participant* parameter cannot have a relationship instance ID. Normally, participant instances do have relationship instance IDs. Because this method creates a new relationship instance, you must make sure that the participant instance does not already have an instance associated with it. To do this, use the `setInstanceId()` method (in the `Participant` class) to set the instance ID to the `INVALID_INSTANCE_ID` constant.

The `create()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the `Relationship` class.

## See also

`addMyChildren()`, `setInstanceId()`

---

## deactivateParticipant()

Deactivates a participant from one or more relationship instances.

### Syntax

```
void deactivateParticipant(String relDefName,  
String partDefName,  
BusObj partData)
```

```
void deactivateParticipant(String  
relDefName,  
String partDefName,  
String partData)
```

```
void deactivateParticipant(String relDefName,  
String partDefName,  
long partData)
```

```
void deactivateParticipant(String relDefName,  
String partDefName,  
int partData)
```

```
void deactivateParticipant(String relDefName,  
String partDefName,  
double partData)
```

```
void deactivateParticipant(String relDefName,  
String partDefName,  
float partData)
```

```
void deactivateParticipant(String relDefName,  
String partDefName,  
boolean partData)
```

```
void deactivateParticipant(Participant participant)
```

### Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition.
<i>partData</i>	Data associated with the participant. Can be one of the following data types: BusObj, String, long, int, double, float, boolean.
<i>participant</i>	Participant to deactivate in the relationship.

### Return values

None.

### Exceptions

RelationshipRuntimeException

### Notes

The `deactivateParticipant()` method deactivates the participant from all instances of *relDefName* where *partData* is associated with *partDefName*. This method does

*not* remove the participant from the relationship tables. Use this method to remove a participant while preserving a record of its existence in the relationship tables.

To view deactivated participants, you can query the relationship tables directly. To find the table names and access information for a given relationship, open the relationship definition using Relationship Designer and choose Advanced Settings from the Edit menu. See “Specifying advanced relationship settings” on page 249 for more information on these settings.

**Attention:** Because `deactivateParticipant()` does not actually remove participant rows from your relationship tables, you should not use this method routinely to delete participants. Doing so can cause your relationship tables to become unnecessarily large.

The `deactivateParticipant()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the `Relationship` class.

## See also

`deleteParticipant()`, `deactivateParticipantByInstance()`, Chapter 7, “Creating relationship definitions,” on page 237, “Transforming with a submap” on page 43

---

## deactivateParticipantByInstance()

Deactivates a participant from a specific relationship instance.

### Syntax

```
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, BusObj partData ] )  
  
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, String partData ] )  
  
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, long partData ] )  
  
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, int partData ] )  
  
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, double partData ] )  
  
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, float partData ] )  
  
void deactivateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [, boolean partData ] )
```

### Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition.
<i>instanceId</i>	ID of the relationship instance to which the participant belongs.
<i>partData</i>	Data associated with the participant. Can be one of the following data types: <code>BusObj</code> , <code>String</code> , <code>long</code> , <code>int</code> , <code>double</code> , <code>float</code> , <code>boolean</code> . This is an optional parameter

## Return values

None.

## Exceptions

RelationshipRuntimeException – See “Handling exceptions” on page 186.

## Notes

The `deactivateParticipantByInstance()` method deactivates the specified participant from the relationship instance that relationship instance ID *instanceID* identifies. However, the method does *not* remove the participant from the relationship tables. Use this method when you want to remove a participant while preserving a record of its existence in the relationship tables.

To view deactivated participants, you can query the relationship tables directly. To find the table names and access information for a given relationship, open the relationship definition using Relationship Designer and choose Advanced Settings from the Edit menu. See “Specifying advanced relationship settings” on page 249 for more information on these settings.

**Attention:** Since `deactivateParticipantByInstance()` does not actually remove participant rows from your relationship tables, you should not use this method routinely to delete participants. Doing so can cause your relationship tables to become unnecessarily large.

The `deactivateParticipantByInstance()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the Relationship class.

## See also

`deleteParticipant()`, `deactivateParticipant()`

---

## deleteParticipant()

Removes a participant instance from one or more relationship instances.

## Syntax

```
void deleteParticipant(String relDefName, String partDefName,
BusObj partData)
void deleteParticipant(String relDefName, String partDefName,
String partData)
void deleteParticipant(String relDefName, String partDefName,
long partData)
void deleteParticipant(String relDefName, String partDefName,
int partData)
void deleteParticipant(String relDefName, String partDefName,
double partData)
void deleteParticipant(String relDefName, String partDefName,
float partData)
void deleteParticipant(String relDefName, String partDefName,
boolean partData)

void deleteParticipant(Participant participant)
```

## Parameters

*relDefName*      Name of the relationship definition.

<i>partDefName</i>	Name of the participant definition.
<i>partData</i>	Data associated with the participant. Can be one of the following data types: BusObj, String, long, int, double, float, boolean.
<i>participant</i>	A Participant instance representing the participant to remove from the relationship.

## Return values

None.

## Exceptions

RelationshipRuntimeException

## Notes

The `deleteParticipant()` method deletes the specified participant from all instances of *relDefName* where *partData* is associated with *partDefName* and deletes it from the underlying relationship tables.

The `deleteParticipant()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the Relationship class.

## See also

`deactivateParticipant()`, `deleteParticipantByInstance()`

---

## deleteParticipantByInstance()

Removes a participant from a specific relationship instance.

### Syntax

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, BusObj partData] )
```

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, String partData] )
```

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, long partData] )
```

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, int partData] )
```

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, double partData] )
```

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, float partData] )
```

```
void deleteParticipantByInstance(String relDefName,
    String partDefName, int instanceId [, boolean partData] )
```



## Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition.
<i>instanceId</i>	ID of the relationship instance to which the participant belongs.
<i>partData</i>	Data associated with the participant. Can be one of the following data types: BusObj, String, long, int, double, float, boolean. This is an optional parameter.

## Return values

None.

## Exceptions

RelationshipRuntimeException

## Notes

The `deleteParticipantByInstance()` method deletes a participant instance from the relationship identified by the *instanceId* relationship instance ID. The method removes the participant from the relationship instance and from the underlying relationship tables.

If you supply the optional *partData* parameter, `deleteParticipantByInstance()` deletes the participant instance *only* if *partData* is the data associated with the *partDefName* participant definition.

The last form of the method accepts a participant instance as the only parameter. The participant instance must contain the relationship definition name, participant definition name, and either the instance ID or the participant data.

The `deleteParticipantByInstance()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the Relationship class.

## See also

`deactivateParticipant()`

---

## getNewID()

Returns the next available relationship instance ID for a relationship, based on the relationship definition name.

## Syntax

```
public static int getNewID(String relDefName)
```

## Parameters

<i>relDefName</i>	Name of the relationship definition.
-------------------	--------------------------------------

## Return values

Returns a relationship instance ID, based on the relationship definition name.

## Exceptions

RelationshipRuntimeException

## Notes

Because the relationship instance ID can be used as the generic ID for the typical IBM WebSphere InterChange Server identity relationships, this new ID can be used as the generic ID for generic-to-generic relationships.

---

## retrieveInstances()

Retrieves only the relationship instance IDs for zero or more relationship instances which contain a given participant instance.

## Syntax

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
BusObj partData)
```

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
String partData)
```

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
long partData)
```

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
int partData)
```

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
double partData)
```

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
float partData)
```

```
int[] retrieveInstances(String relDefName,  
String partDefName,  
boolean partData)
```

```
int[] retrieveInstances(String relDefName,  
String[] partDefList,  
BusObj partData)
```

```
int[] retrieveInstances(String relDefName,  
String[] partDefList,  
String partData)
```

```
int[] retrieveInstances(String relDefName,  
String[] partDefList,  
long partData)
```

```
int[] retrieveInstances(String relDefName,  
String[] partDefList,  
int partData)
```

```
int[] retrieveInstances(String relDefName,  
String[] partDefList,  
double partData)
```

```

int[] retrieveInstances(String relDefName,
String[] partDefList,
float partData)

int[] retrieveInstances(String relDefName,
String[] partDefList,
boolean partData)

int[] retrieveInstances(String relDefName, BusObj partData)
int[] retrieveInstances(String relDefName, String partData)
int[] retrieveInstances(String relDefName, long partData)
int[] retrieveInstances(String relDefName, int partData)
int[] retrieveInstances(String relDefName, double partData)
int[] retrieveInstances(String relDefName, float partData)
int[] retrieveInstances(String relDefName, boolean partData)

```

## Parameters

*relDefName* Name of the relationship definition.

*partDefName* Name of the participant definition.

*partDefList* List of participant definitions.

*partData* Data to associate with the participant. Can be one of the following data types: BusObj, String, long, int, double, float, boolean.

## Return values

Returns an array of integers that are the instance IDs of relationships containing the participant.

## Exceptions

RelationshipRuntimeException

## Notes

The `retrieveInstances()` method implements a lookup relationship in an inbound map. It obtains the relationship instance IDs from the relationship table that are associated with the specified participant instances (*partDefList* and *partData* or only *partData*). The method retrieves *only* those attributes that are associated with the *relDefName* relationship definition. It does *not* fill in any of the other attributes in the business object. Attributes associated with the relationship definition typically are the key attributes and any others that you explicitly select. See Chapter 7, “Creating relationship definitions,” on page 237 for more information on relationship definitions.

If `retrieveInstances()` does not find a relationship instance for the specified data, it does *not* raise an exception. Absence of data in the relationship table does not mean that the lookup was performed improperly. If you want to raise an exception when `retrieveInstances()` does not find a value, you must check the value of the instance IDs that the method returns and explicitly raise a `MapFailureException` if the value is null.

The `retrieveInstances()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the Relationship class.

## See also

`addMyChildren()`, `deactivateParticipant()`, `deleteParticipant()`,  
`retrieveParticipants()`

“Customizing map transformations for a lookup relationship” on page 261

---

## retrieveParticipants()

Retrieves zero or more participants from a relationship instance.

### Syntax

```
Participant[] retrieveParticipants(String relDefName,  
    String partDefName, int instanceId)
```

```
Participant[] retrieveParticipants(String relDefName,  
    String[] partDefList, int instanceId)
```

```
Participant[] retrieveParticipants(String relDefName,  
    int instanceId)
```

### Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition.
<i>partDefList</i>	List of participant definitions.
<i>instanceId</i>	The relationship instance ID of the relationship instance to which the participant belongs.

### Return values

Returns an array of Participant instances.

### Exceptions

`RelationshipRuntimeException`

### Notes

The `retrieveParticipants()` method implements a lookup relationship in an outbound map. It obtains the participant instances from the relationship table that are associated with the specified *instanceId* relationship instance ID. The method retrieves *only* those attributes that are associated with the *relDefName* relationship definition. It does *not* fill in any of the other attributes in the business object. Attributes associated with the relationship definition typically are the key attributes and any others that you explicitly select. See Chapter 7, “Creating relationship definitions,” on page 237 for more information on relationship definitions.

If `retrieveParticipants()` raises the `RelationshipRuntimeException` if it receives a null-valued *instanceId*. If you are not guaranteed that the `retrieveInstances()` method has returned a matching instance ID, check the value of *instanceId* for a null value *before* the call to `retrieveParticipants()`.

The `retrieveParticipants()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the `Relationship` class.

## See also

`addMyChildren()`, `deactivateParticipant()`, `deleteParticipant()`,  
`retrieveInstances()`

“Customizing map transformations for a lookup relationship” on page 261

---

## updateParticipant()

Updates a participant in one or more relationship instances.

### Syntax

```
void updateParticipant(String relDefName, String partDefName,  
BusObj partData)
```

### Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition that participates in the <i>relDefName</i> relationship.
<i>partData</i>	Data to associate with the participant. Can be one of the following data types: <code>BusObj</code> .

### Return values

None.

### Exceptions

`RelationshipRuntimeException`

### Notes

The `updateParticipant()` method updates *partData* in instances of *relDefName* where *partData* is associated with *partDefName*. This method updates the non-key attributes of the business object that is associated with the specified participant. Only the attributes that are associated with the relationship definition are updated.

The `updateParticipant()` method updates all participant instances in the *relDefName* relationship that have:

- A participant definition of *partDefName*
- Key value(s) that matches the key value(s) of the *partData* business object

This method updates the non-key attributes of the participant instances with the values in the *partData* business object. Only the attributes that are associated with the relationship definition are updated.

To modify a key attribute or a participant type that is *not* a business object (such as `String`, `long`, `int`, `double`, `float`, or `boolean`), you must first delete the participant using `deleteParticipant()` or `deactivateParticipant()` and then add a new participant using `addMyChildren()`.

The `updateParticipant()` method is a class method declared as static. You can call this method from an existing relationship instance or by referencing the `Relationship` class.

## See also

`deleteParticipant()`, `deactivateParticipant()`, `addMyChildren()`

---

## updateParticipantByInstance()

Updates a participant in a specific relationship instance.

### Syntax

To update a participant in a specific relationship instance:

```
void updateParticipantByInstance(String relDefName,  
    String partDefName, int instanceId [ , BusObj partData ] )
```

```
void updateParticipantByInstance(Participant participant)
```

### Parameters

<i>relDefName</i>	Name of the relationship definition.
<i>partDefName</i>	Name of the participant definition.
<i>instanceId</i>	The relationship instance ID that identifies the relationship to which the participant belongs.
<i>partData</i>	Data to associate with the participant. Can be one of the following data types: <code>BusObj</code> . This parameter is optional.
<i>participant</i>	Participant to update in the relationship.

### Return values

None.

### Exceptions

`RelationshipRuntimeException`

### Notes

The `updateParticipantByInstance()` method updates the non-key attributes of the business object associated with the specified participant. Only the attributes that are associated with the relationship definition are updated.

To modify a key attribute or a participant type that is not a business object (such as `String`, `long`, `int`, `double`, `float`, or `boolean`), you must first delete the participant using `deleteParticipant()` or `deactivateParticipant()` and then add a new participant using `addMyChildren()`.

The `updateParticipantByInstance()` method is a class method declared as `static`. You can call this method from an existing relationship instance or by referencing the `Relationship` class.

## See also

`deleteParticipant()`, `deactivateParticipant()`, `addMyChildren()`

---

## Deprecated methods

Some methods in the Relationship class have been moved to the IdentityRelationship class. These *deprecated methods* will not generate errors, but CrossWorlds recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 146 lists the deprecated methods for the Relationship class.

*Table 146. Deprecated methods, Relationship class*

<b>Former method</b>	<b>Replacement</b>
addMyChildren()	addMyChildren() in the IdentityRelationship class
deleteMyChildren()	deleteMyChildren() in the IdentityRelationship class
maintainCompositeRelationship()	maintainCompositeRelationship() in the IdentityRelationship class
maintainSimpleIdentityRelationship()	maintainSimpleIdentityRelationship() in the IdentityRelationship class
updateMyChildren()	updateMyChildren() in the IdentityRelationship class





---

## Chapter 26. UserStoredProcedureParam class

The UserStoredProcedureParam class provides methods for handling argument values to stored procedures, which you execute on the relationship database. A UserStoredProcedureParam object describes a single parameter for a stored procedure.

**Important:** The UserStoredProcedureParam class and its methods are supported for backward compatibility *only*. These *deprecated methods* will not generate errors, but you should avoid using them and migrate existing code to the new methods. The deprecated methods might be removed in a future release. In new map development, use the CwDBStoredProcedureParam class and its methods to provide arguments to a stored procedure.

Table 147 summarizes the methods in the UserStoredProcedureParam class.

Table 147. UserStoredProcedureParam method summary

Method	Description	Page
UserStoredProcedureParam()	Constructs a new instance of UserStoredProcedureParam that holds argument information for the parameter of a stored procedure.	487
getParamDataTypeJavaObj()	Retrieves the data type of this stored-procedure parameter as a Java Object, such as Integer, Double, or String.	488
getParamDataTypeJDBC()	Retrieves the data type of this stored-procedure parameter as an integer JDBC data type.	489
getParamIndex()	Retrieves the index position of this stored-procedure parameter.	489
getParamIOType()	Retrieves the in/out parameter type for this stored-procedure parameter.	490
getParamName()	Retrieves the name of this stored-procedure parameter.	491
getParamValue()	Retrieves the value of this stored-procedure parameter.	491
setParamDataTypeJavaObj()	Sets the data type as a Java Object for this stored-procedure parameter.	492
setParamDataTypeJDBC()	Sets the data type as a JDBC data type for this stored-procedure parameter.	492
setParamIndex()	Sets the index position of this stored-procedure parameter.	493
setParamIOType()	Sets the in/out parameter type of this stored-procedure parameter.	493
setParamName()	Sets the name of this stored-procedure parameter.	494
setParamValue()	Sets the value of this stored-procedure parameter.	494

---

### UserStoredProcedureParam()

Constructs a new instance of UserStoredProcedureParam that holds argument information for the parameter of a stored procedure.

## Syntax

```
UserStoredProcedureParam(int paramIndex, String paramType,  
    Object paramValue, String ParamIOType, String paramName)
```

## Parameters

<i>paramIndex</i>	The index position of the associated parameter in the declaration of the stored procedure. Index numbering begins with one (1).
<i>paramType</i>	The data type (as a Java Object) of the associated parameter.
<i>paramValue</i>	The argument value to send to the stored procedure.
<i>ParamIOType</i>	The in/out type of the associated parameter. Valid types are: "IN" - parameter value is <i>input only</i> . "INOUT" - parameter value is <i>input and output</i> . "OUT" - parameter value is <i>output only</i> .
<i>paramName</i>	The name of the argument, to be used in later statements that build the Vector array.

## Return values

Returns a new UserStoredProcedureParam object to hold the argument information for the argument at position *argIndex* in the declaration of the stored procedure.

## Exceptions

DtpConnectionException – If a parameter is invalid.

---

## getParamDataTypeJavaObj()

Retrieves the data type of this stored-procedure parameter as a Java Object, such as Integer, Double, or String.

## Syntax

```
String getParamDataTypeJavaObj()
```

## Parameters

None.

## Return values

Returns the data type of the associated UserStoredProcedureParam parameter as a Java Object.

## Exceptions

None.

## Notes

A Java Object is one of two representations of the parameter data type stored in the UserStoredProcedureParam object. Use getParamDataTypeJavaObj() to obtain the Java Object data type, you should work with the Java Object data type because:

- For IN (and INOUT) parameters, you *must* provide the parameter value as a Java Object. Therefore, providing the parameter data type as a Java Object is more consistent.

- The `execStoredProcedure()` method sends parameters in a `Vector` parameter array. The `Vector` object can contain only elements that are Java Objects.

## See also

`getParamDataTypeJDBC()`, `setParamDataTypeJavaObj()`

---

## getParamDataTypeJDBC()

Retrieves the data type of this stored-procedure parameter as an integer JDBC data type.

### Syntax

```
int getParamDataTypeJDBC()
```

### Parameters

None.

### Return values

Returns the data type of the associated `UserStoredProcedureParam` parameter as a JDBC data type.

### Exceptions

None.

### Notes

The JDBC data type is one of two representations of the parameter data type stored in the `UserStoredProcedureParam` object. JDBC data types are integer values and include the following:

- `java.sql.Types.INTEGER`
- `java.sql.Types.VARCHAR`
- `java.sql.Types.DOUBLE`
- `java.sql.Types.DATE`

These data types are defined in `java.sql.Types`.

**Recommendation:** You should use the Java Object data type instead of the JDBC data type. However, the Mapping API uses the JDBC internally so you can obtain its value from the `UserStoredProcedureParam` object with `getParamDataTypeJDBC()`.

## See also

`getParamDataTypeJavaObj()`, `setParamDataTypeJDBC()`

---

## getParamIndex()

Retrieves the index position of this stored-procedure parameter.

### Syntax

```
int getParamIndex()
```

## Parameters

None.

## Return values

Returns the index position of the associated `UserStoredProcedureParam` parameter.

## Exceptions

None.

## Notes

The index position of a stored-procedure parameter is its position in the parameter list of the stored-procedure declaration. The first parameter has an index position of one (1). The index position does *not* refer to literal parameters that might be supplied to the stored procedure.

## See also

`setParamIndex()`

---

## getParamIOType()

Retrieves the in/out parameter type for this stored-procedure parameter.

## Syntax

```
String getParamIOType()
```

## Parameters

None.

## Return values

Returns the in/out type of the associated `UserStoredProcedureParam` parameter.

## Exceptions

None.

## Notes

The in/out parameter type indicates how the stored procedure uses the parameter. It can be the string representation of one of the following:

- IN parameter  
An IN parameter is *input only*; that is, the stored procedure accepts its value as input but does *not* use the parameter to return a value. The `getParamIOType()` returns the in/out parameter type as "IN".
- INOUT parameter  
An INOUT parameter is *input and output*; that is, the stored procedure accepts its value as input and also uses the parameter to return a value. The `getParamIOType()` returns the in/out parameter type as "INOUT".
- OUT parameter  
An OUT parameter is *output only*; that is, the stored procedure does *not* read its value as input but does use the parameter to return a value. The `getParamIOType()` returns the in/out parameter type as "OUT".

## See also

`setParamIOType()`

---

## getParamName()

Retrieves the name of this stored-procedure parameter.

### Syntax

```
String getParamName()
```

### Parameters

None.

### Return values

Returns the name of the parameter from the associated `UserStoredProcedureParam` object.

### Exceptions

None.

### Notes

The name of the parameter is informational only. It is used only for error messages and debugging. The parameter name is not needed to access the stored-procedure parameter because stored procedures are accessed by their index position in the stored-procedure declaration.

## See also

`setParamName()`

---

## getParamValue()

Retrieves the value of this stored-procedure parameter.

### Syntax

```
Object getParamValue()
```

### Parameters

None.

### Return values

Returns the value of the associated `UserStoredProcedureParam` 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

`setParamValue()`

---

## setParamDataTypeJavaObj()

Sets the data type as a Java Object for this stored-procedure parameter.

### Syntax

```
void setParamDataTypeJavaObj(String paramDataType)
```

### Parameters

*paramDataType* The data type of the parameter as a Java Object.

### Exceptions

`DtpConnectionException` – If the input data type is not supported.

### Notes

A Java Object is one of two representations of the parameter data type stored in the `UserStoredProcedureParam` object. Use `setParamDataTypeJavaObj()` to set the data type as a Java Object. You should work with the Java Object data type because:

- For IN (and INOUT) parameters, you *must* provide the parameter value as a Java Object. Therefore, providing the parameter data type as a Java Object is more consistent.
- The `execStoredProcedure()` method sends parameters in a Vector parameter array. The Vector object can contain only elements that are Java Objects.

## See also

`getParamDataTypeJavaObj()`, `setParamDataTypeJDBC()`

---

## setParamDataTypeJDBC()

Sets the data type as a JDBC data type for this stored-procedure parameter.

### Syntax

```
void setParamDataTypeJDBC(int paramDataType)
```

### Parameters

*paramDataType* The data type of the parameter as a JDBC type.

### Exceptions

`DtpConnectionException` – If the input data type is not supported.

### Notes

Every `UserStoredProcedureParam` object contains two representations of its data type: Java Object and JDBC data type. You should use the Java Object data type because:

- For IN (and INOUT) parameters, you *must* provide the parameter value as a Java Object. Therefore, providing the parameter data type as a Java Object is more consistent.
- The `execStoredProcedure()` method sends parameters in a Vector parameter array. The Vector object can contain only elements that are Java Objects.

## See also

`getParamDataTypeJDBC()`, `getParamDataTypeJavaObj()`

---

## setParamIndex()

Sets the index position of this stored-procedure parameter.

### Syntax

```
void setParamIndex(int paramIndex)
```

### Parameters

*paramIndex*      The index position of the stored-procedure parameter

### Notes

The index position of a stored-procedure parameter is its position in the parameter list of the stored-procedure declaration. The first parameter has an index position of one (1). The index position does *not* refer to literal parameters that might be supplied to the stored procedure.

## See also

`getParamIndex()`

---

## setParamIOType()

Sets the in/out parameter type of this stored-procedure parameter.

### Syntax

```
void setParamIOType(String paramIOType)
```

### Parameters

*paramIOType*      The I/O type of the stored-procedure parameter

### Notes

The in/out parameter type indicates how the stored procedure uses the parameter. It can be any of the following:

- IN parameter  
An IN parameter is *input only*; that is, the stored procedure accepts its value as input but does *not* use the parameter to return a value. For an IN parameter, set the in/out parameter type to "IN".
- INOUT parameter  
An INOUT parameter is *input and output*; that is, the stored procedure accepts its value as input and also uses the parameter to return a value. For an INOUT parameter, set the in/out parameter type to "INOUT".
- OUT parameter

An OUT parameter is *output only*; that is, the stored procedure does *not* read its value as input but does use the parameter to return a value. For an OUT parameter, set the in/out parameter type to “OUT”.

## See also

`getParamIOType()`

---

## setParamName()

Sets the name of this stored-procedure parameter.

### Syntax

```
void setParamName(String paramName)
```

### Parameters

*paramName*      The name of the stored-procedure parameter

### Notes

The name of the parameter is informational only. It is used only for error messages and debugging. The parameter name is not needed to access the stored-procedure parameter because stored procedures are accessed by their index position in the stored-procedure declaration.

## See also

`getParamName()`

---

## setParamValue()

Sets the value of this stored-procedure parameter.

### Syntax

```
void setParamValue(Object paramValue)
```

### Parameters

*paramValue*      The value of the stored-procedure parameter. The value must be a Java Object (such as Integer, Double, or String).

### Notes

You must set the parameter value as a Java Object.

## See also

`getParamValue()`



---

## Part 4. Appendixes



---

## Appendix A. Message files

Each map can have an associated message file. The *message file* contains the text for the map'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, called *parameters*.

The methods that generate map messages provide two ways of generating the message text that a user sees. The coding of the method call can:

- Include the text of the message.
- Contain a reference to message text that is contained in an external message file.

It is generally a better practice for a map to refer to a message file than to generate the text itself, for ease of maintenance, administration, and internationalization.

This chapter describes message files, how they work, and how to set them up. It covers the following topics:

---

"Message location"	497
"Format for map messages" on page 500	500
"Message parameters" on page 501	501
"Maintaining the files" on page 502	502
"Operations that use message files" on page 502	502

---

---

### Message location

All message file are located in the following directory of the IBM WebSphere InterChange Server product directory:

DLMs\messages

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

Three types of message files can be used to generate messages for a map:

- A map-specific message file, *mapName\_locale.txt* where *mapName* corresponds to the name of the map and *locale* corresponds to the locale that the map is defined in.

Map messages appear in the Messages tab of Map Designer and are stored as part of the map definition in the repository. When you compile the map, Map Designer extracts the message content and creates (or updates) the message file for run-time use. The name of the message file has the following format:

*MapName\_locale.txt*

**Example:** For the LegacyAddress\_to\_CwAddress map, if it is created in an English locale in the United States, Map Designer creates the message file called LegacyAddress\_to\_CwAddress\_en\_US.txt and places it in the ProjectName\Maps\Messages directory. After the map is deployed to InterChange Server, it will be placed in the DLMs\messages directory.

- The `UserMapMessages.txt` message file  
To this file, you can add new message numbers that fall into a “safe” range, as defined by WebSphere Business Integration (see Table 148). For example, if you create a message for an Oracle map, you would assign the message a number between 6101 and 6200. You can also use a message number that is already defined in the WebSphere Business Integration generic message file (`CwMapMessages.txt`, described next) and change the existing message text to text of your choice. Since the `UserMapMessages.txt` file is searched *before* the WebSphere InterChange Server message file, your additions override those messages.
- The WebSphere InterChange Server generic message file, `CwMapMessages.txt` (which WebSphere InterChange Server provides).  
If your map does *not* reference one of the other two message files, it must reference this one. Table 148 lists the message numbers that WebSphere InterChange Server has assigned and that are contained in the generic message file.  
**Attention:** Do *not* change the contents of the WebSphere InterChange Server generic message file `CwMapMessages.txt`! Make changes to a generic message by copying it into the `UserMapMessages.txt` message file and customizing it.

These files range from map-specific to general purpose. Messages that can be used by any map are located in a generic file, provided by WebSphere InterChange Server. The other two files provide you with the option to customize messages for your maps, as needed.

**Important:** InterChange Server reads the `UserMapMessages.txt` and `CwMapMessages.txt` files into memory when it starts up. If you make changes to `UserMapMessages.txt`, you *must* restart InterChange Server for these changes to be available to maps.

Table 148. `CwMapMessages.txt` messages

Message number	Message text	Message usage
5000	Mapping - Value of the primary key in the source object is null. Map execution stopped.	Used if the primary key of the source object is null. The check for the source primary key = null should always be performed before any of the relationship methods are called that are based on the source object’s primary key. If the key is null, the error should display and the map should stop execution.
5001	Mapping - RelationshipRuntimeException. Map execution stopped.	Used if RelationshipRuntimeException is caught in one of the following: <ul style="list-style-type: none"> <li>• Function blocks <ul style="list-style-type: none"> <li>– General/APIs/Identity Relationship/Maintain Simple Identity Relationship</li> <li>– General/APIs/Identity Relationship/Maintain Composite Relationship</li> </ul> </li> <li>• Mapping APIs <ul style="list-style-type: none"> <li>– maintainSimpleIdentityRelationship()</li> <li>– maintainCompositeRelationship()</li> </ul> </li> </ul>

Table 148. CwMapMessages.txt messages (continued)

Message number	Message text	Message usage
5002	Mapping - CxMissingIDException. Map execution stopped.	Used if CxMissingIDException is caught in one of the following: <ul style="list-style-type: none"> <li>• Function blocks <ul style="list-style-type: none"> <li>– General/APIs/Identity Relationship/Maintain Simple Identity Relationship</li> <li>– General/APIs/Identity Relationship/Maintain Composite Relationship</li> </ul> </li> <li>• Mapping APIs <ul style="list-style-type: none"> <li>– maintainSimpleIdentityRelationship()</li> <li>– maintainCompositeRelationship()</li> </ul> </li> </ul>
5003	Mapping - Data in the {1} attribute is missing.	Used when the source attribute is null before using the function block Foreign Key Lookup (foreignKeyLookup()) or Foreign Key Cross-Reference (foreignKeyXref()). The check for the source attribute = null should always be performed before these relationship methods are called. If the key is null, the error should be displayed and the map might stop execution.
5007	Mapping - ForeignKeyLookup() of '{1}' with Source Value of '{2}' failed for the '{3}' relationship and '{4}' participant on Initiator '{5}'. Map execution stopped.	Used if the destination attribute is null after using the function block Foreign Key Lookup (foreignKeyLookup()). Map has to stop execution.
5008	Mapping - ForeignKeyLookup() of '{1}' with Source Value of '{2}' failed for the '{3}' relationship and '{4}' participant on Initiator '{5}'. Map execution continued.	Used if the destination attribute is null after using the function block Foreign Key Lookup (foreignKeyLookup()). Map has to continue execution.
5009	Mapping - ForeignKeyXref() of '{1}' with Source Value of '{2}' failed for the '{3}' relationship and '{4}' participant on Initiator '{5}'. Map execution stopped.	Used if the destination attribute is null after using the function block Foreign Key Cross-Reference (foreignKeyXref()). Map has to stop execution.

When a map references a message number, the message files are searched in the following order:

1. The map-specific message file *mapName\_locale.txt* where *mapName* corresponds to the name of the map, is searched.
2. The file *UserMapMessages.txt* is searched.
3. The WebSphere InterChange Server generic message *CwMapMessages.txt* is searched.

Table 149 shows code examples that demonstrate situations in which each of the messages in the *CwMapMessages.txt* file might be used.

Table 149. Code Examples for CwMapMessages.txt Messages

Message number	Code example
5000	<pre>ObjContract.setVerb(ObjSAP_Contract.getVerb()); if (ObjSAP_Contract.get("ContractId") == null) { logError(5000); throw new MapFailureException("Data in the primary key is missing"); }</pre>

Table 149. Code Examples for CwMapMessages.txt Messages (continued)

Message number	Code example
5001	<pre>try { IdentityRelationship.maintainSimpleIdentityRelationship( "Contract", "SAPCntr", ObjSAP_Contract, ObjContract, cwExecCtx); } catch (RelationshipRuntimeException e1) { logError(5001); throw new MapFailureException( "RelationshipRuntimeException"); } catch (CxMissingIDException e2) { logError(5002); throw new MapFailureException("CxMissingIDException"); }</pre>
5002	See code example above.
5003	<pre>if (ObjSAP_Contract.get("CustomerId") == null) { logError(5003, "CustomerId"); throw new MapFailureException("CustomerId is null"); }</pre>
5007	<pre>try { IdentityRelationship.foreignKeyLookup ("Customer", "OracCust", ObjOracle_OrderImport, "customer_id", ObjOrder, "CustomerId", cwExecCtx); } catch (RelationshipRuntimeException e) { logWarning(e.toString()); } if (ObjOracle_OrderImport.get("customer_id") == null) { logError(5007, "customer_id", "CustomerId", "Customer", "OracCust", strInitiator); throw new MapFailureException( "foreignKeyLookup() failed."); } }</pre>
5008	<pre>try { IdentityRelationship.foreignKeyLookup ("Customer", "OracCust", ObjOracle_OrderImport, "customer_id", ObjOrder, "CustomerId", cwExecCtx); } catch (RelationshipRuntimeException e) { logWarning(e.toString()); } if (ObjOracle_OrderImport.get("customer_id") == null) { logError(5008, "customer_id", "CustomerId", "Customer", "OracCust", strInitiator); } }</pre>
5009	<pre>try { IdentityRelationship.foreignKeyXref ("Customer", "OracCust", "CWCust", ObjOracle_OrderImport, "customer_id", ObjOrder, "CustomerId", cwExecCtx); } catch (RelationshipRuntimeException e) { logWarning(e.toString()); } if (ObjOracle_OrderImport.get("customer_id") == null( { logError(5009, "customer_id", "CustomerId", "Customer", "OracCust", strInitiator); throw new MapFailureException( "foreignKeyXref() failed."); }</pre>

## Format for map messages

To ensure consistency of messages, WebSphere InterChange Server has developed a message format. This section describes that format, including:

- “Message format” on page 501
- “Message parameters” on page 501
- “Comments” on page 502

**Note:** The map-specific message file should be modified from the message tab in Map Designer and should not be modified directly. Map Designer will overwrite any custom modification in the map-specific message file with the messages saved in the map. However, for the message files `UserMapMessages.txt` and `CwMapMessages.txt`, it is safe to modify the file directly.

## Message format

The format for each message is:

```
MessageNum  
Message
```

The message number (*MessageNum*) and the message itself (*Message*) must be on different lines, with a carriage return at the end of each line.

**Example:** A map's messages might include a message identified as number 23, whose text includes two placeholder variables, marked as {1} and {2}, as shown in Figure 140..

```
23  
Customer ID {1} could not be changed: {2}
```

Figure 140. Sample Message

## Message parameters

When the map calls a method that displays a particular message, it passes to 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.

It is not necessary to write separate messages for each possible situation. Instead, use parameters to represent values that change at run time. 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.
```

**Example:** Consider message 23 in Figure 140 again. When the map wants to display or log this message, it passes to the appropriate method the identifying number of the message (23) and two additional parameters:

- Parameter 1 becomes the customer ID number (6701)
- Parameter 2 becomes 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
```

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 customer ID and explanatory text.

## Comments

Precede each comment line in a message file with a pound sign (#).

**Example:** A comment might look like this:

```
# Message file for the Address business object map.
```

**Recommendation:** It is good practice to start the file with a series of comment lines to form a short header. Include in the header data the name of the map and such information as the file creator and file creation date.

---

## Maintaining the files

At a user site, an administrator might set up a procedure for filtering map messages and notifying someone who can resolve problems, by e-mail or e-mail pager. This means that the error numbers and the meanings associated with the numbers must remain the same after the first release of a map.

**Recommendation:** You can change the text associated with an error number, but you should avoid changing the meaning of the text or reassigning error numbers. If you do change the meanings associated with error numbers, you should document the change and notify users of the map.

---

## Operations that use message files

Message files hold text for messages used in several types of operations. Table 9 on page 23 lists the types of operations that use message files and the methods of the BaseDLM class that perform those operations.

Table 150. Message-generating operations

Operation	Function block	Method
Raising exceptions	General/APIs/Maps/Exception/ Raise Map Exception	raiseException()
Logging	General/Logging and Tracing/Log Information ID	logInfo()
	General/Logging and Tracing/Log error ID	logError()
	General/Logging and Tracing/Log warning ID	logWarning()
Tracing	General/Logging and Tracing/Trace/Trace on Level	trace()

This section describes message-generating operations that affect map execution.

### Raising exceptions

The `raiseException()` method has several forms. One commonly used syntax is:

```
raiseException(String exceptionType,  
               int messageNum, String param[,...])
```

With this syntax, you can have from one to three *param* String parameters. Thus, there can be up to five comma-separated parameters in a call to `raiseException()`.



This example raises a new exception, using message number 23, and passes in two parameters to the message, the customer ID value and a string:

```
raiseException(AttributeException, 23,  
    fromCustomer.getString("CustomerID"),  
    "greater than maximum length");
```

Figure 140 shows the text for message 23 as it appears in the message file.

## Logging messages

A map can log a message whenever something occurs that might be of interest to an administrator. To log a message, a map uses the `logInfo()`, `logWarning()`, and `logError()` methods of the `BasedLM` class. Each method is associated with a different message severity level.

### Severity levels

To log a message, you must call the method associated with the message's severity level. Table 151 lists the severity levels and their associated methods.

Table 151. Message levels

Severity level	Method	Description
Info	<code>logInfo()</code>	Informational only. The user does not need to take action.
Warning	<code>logWarning()</code>	Represents information about a problem. Do not use this level for problems that the user must resolve.
Error	<code>logError()</code>	Indicates a serious problem that the user needs to investigate.

### Using a message file

Every map has at least one message file associated with it. If a map does not use custom messages, its messages come from the system map message file, `CwMapMessages.txt`. If a map uses customized messages, it has a map-specific message file (which is generated from the messages entered in the Messages tab of Map Designer). For more information, see "Message location" on page 497.

When a map logs an error, the text of the error message comes from the map's message file.

**Example:** The following example logs an error message whose text is contained in the map's message file. The text of error message 10 appears as follows in the message file:

```
10  
Credit report error for {1}, {2}.
```

The code to log the message looks like this:

```
logError(10, customer.get("LName"), customer.get("FName"));
```

When the `logError()` method executes, the text for message 10 is written to the log file, with the customer's last name and first name substituted for parameters 1 and 2.

**Example:** The logged message for a customer named John Davidson looks like this:  
Credit report error for Davidson, John.

## Principles of good message logging

When creating messages, be sensitive to the way that administrators use the logging feature.

**Assigning severity levels:** It is important to be precise when assigning error levels to messages. The IBM system e-mail 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. Administrators use this IBM system e-mail notification feature, and they additionally might link it to an e-mail 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, once 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 map so that when a map instance 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 map's trace level to a number higher than zero. You can send trace messages and log messages to different files.

You can add trace messages to a map to report operations that are specific to that map. These are some types of information that the map can write to the trace file:

- Key values of a business object at the point that the map begins or ends a particular transformation step.
- The decision to take a particular branch in the execution path.

### 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.

**Recommendations:** 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.
- 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

**Example:** 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  
Begin transformation on {1} attribute: value = {2}
```

The method call obtains the value of the attribute LName, then uses the value to replace the parameter in the message. The code appears in the map as follows, and the message appears when the user sets tracing to level 3:

```
trace(3, 20, "LName", customer.get("LName"));
```

### Setting the trace level

Figure 141 shows the General tab of the Map Properties dialog in Map Designer. (For information on how to display the Map Properties dialog, see “Specifying map property information” on page 58.) Notice that you can set the trace level for

trace messages in this dialog.

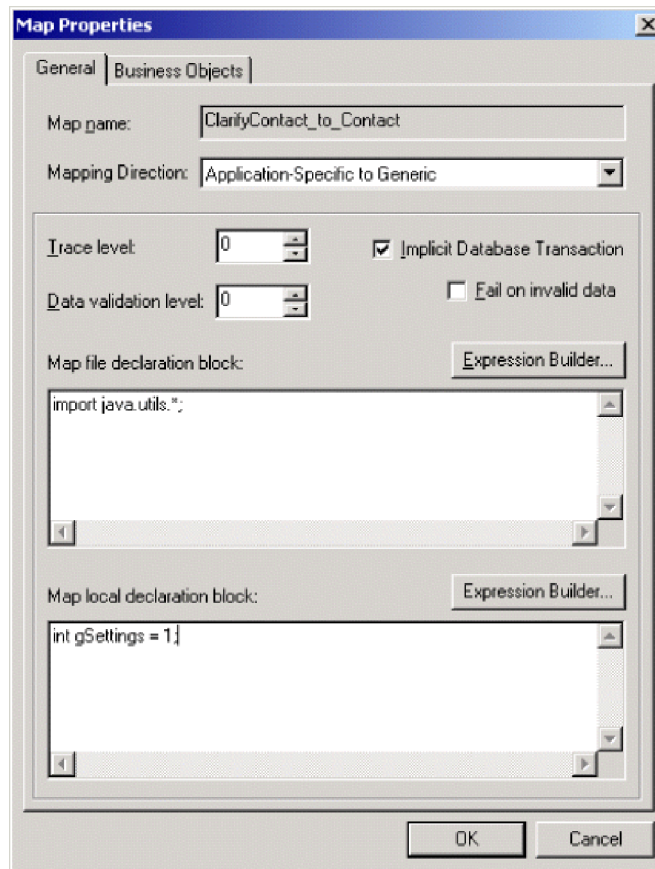


Figure 141. Trace level for a map

As the map developer, you create the levels for which map-generated tracing can be requested, as described in “Assigning trace levels” on page 504.

**Note:** If you change the trace level for an activated map, you must stop and restart the map before the new trace level takes effect. Use the Component menu of System Manager to stop and start a map.

By setting the trace level in the Map Properties dialog of Map Designer, you set it for *all* map instances based on this map definition. You can also set the trace level for all map instances from the Map Properties window of System Manager.

---

## Appendix B. Attribute properties

Table 152 lists the properties for attributes of business object definitions.

Table 152. Attribute Properties

Property	Description
Name	A name that describes what type of data the attribute contains. The name must be less than or equal to 80 alphanumeric characters and underscores. It cannot contain spaces or certain punctuation symbols, such as a period, a left brace ({}), a right brace (}), a single quotation mark, or a double quotation mark.
Type	The data type of the attribute. Basic types include String, Boolean, Double, Float, Integer, and Date. If the attribute references a child business object, specify the name of a child business object definition. Attributes that reference child business objects are called compound attributes.
IsKey	A boolean value, true or false, specifying whether this is a key attribute. Key attributes uniquely identify a business object created from the definition. Each business object definition has at least one key attribute.
IsForeignKey	A boolean value, true or false, specifying whether this is a foreign key attribute.
MaxLength	An integer representing the maximum number of bytes the attribute can contain. To specify no limit, enter zero (0).
AppSpecificInfo	A string that provides information about the attribute for a particular application, such as the name of a field in a table or form that corresponds to the attribute. Connectors use this information when processing the object.
DefaultValue	The value to assign to this attribute if there is no runtime value.
IsRequired	A boolean value, true or false, specifying whether a value for this attribute is required to create a business object.
ContainedObjectVersion	The version number of the child business object definition. IBM WebSphere System Manager displays this value under the name Type Version.
Relationship	The relationship between the parent business object and the child business object. In the current release, the only valid relationship is Containment.
Cardinality	The number of child business objects that this attribute references. If the attribute references only one child business object, the value is 1. If the attribute can reference many child business objects, the value is a literal n.



---

## 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.

System Manager includes software developed by the Eclipse Project (<http://www.eclipse.org/>).



WebSphere InterChange Server V4.3.0 and WebSphere InterChange Server  
Intgration Toolset V4.3.0

---

# Index

## Special characters

.bo file extension 13, 89, 90, 94  
.class file extension 13, 84  
.cwm file extension 13, 50, 57  
.dln file extension xv, 80  
.jar file extension 79  
.java file extension 13, 84, 85, 86  
.txt file extension 13, 497

## A

Access client 87, 192, 265  
ACCESS\_REQUEST calling context 190, 192, 265  
    Create verb and 265, 288, 292, 296  
    Delete verb and 266, 288, 297  
    foreignKeyXref() and 291, 443  
    getOriginalRequestBO() and 459  
    maintainChildVerb() and 288, 446  
    maintainCompositeRelationship() and 276, 447  
    maintainSimpleIdentityRelationship() and 265, 450  
    original-request business object 192, 459  
    Retrieve verb and 266, 288, 292  
    retrieving 458  
    setting to 461  
    testing with 96, 99  
    Update verb and 266, 288, 292  
ACCESS\_RESPONSE calling context 190, 192, 265  
    foreignKeyXref() and 294, 443  
    getOriginalRequestBO() and 460  
    maintainCompositeRelationship() and 276, 447  
    maintainSimpleIdentityRelationship() and 272, 450  
    original-request business object 192, 273, 460  
    retrieving 458  
    setting to 461  
    updateMyChildren() and 284  
Activity Editor 103  
    accessing 28, 38, 41, 43, 46, 104  
    Add Comment 110  
    Add Description 110, 155  
    Add Label 110  
    Add To do 110  
    Add To My Collection 110  
    Bidirectional functionality 162  
    Check for Unmatched Delimiters 83  
    connection links 113  
    Content window 106  
    Context menu 110  
    Cross-Reference transformation 104  
    Custom transformation 48  
    Design mode 105, 107  
    Document Display Area 105  
    Edit menu 109  
    example of using 144, 148, 157  
    File menu 108  
    function blocks 112, 116, 282  
    Graphical view 104, 105  
    grouping components 115  
    Help menu 110  
    Java view 104, 107  
    Join transformation 41, 104  
Activity Editor (*continued*)  
    keyboard shortcuts 108  
    Label 114  
    layout 104  
    Library window 105  
    main menus 108  
    main views 104  
    New Constant 110, 115, 154  
    Preferences dialog 49  
    Properties window 106  
    Quick view mode 106, 107  
    Resize label 114  
    saving an activity 148, 156  
    Set Value transformation 38, 104  
    Split transformation 43, 104  
    starting 103  
    Status bar 111  
    Submap transformation 46, 104  
    Title Bar 104  
    toolbars 110  
    Tools menu 109  
    using function blocks in 112  
    using Web services in 159  
    View menu 109  
addDays() method 179, 402  
addElement() method 348  
addMyChildren() method 187, 296, 437, 464, 485  
addParticipant() method 296, 472  
addWeekdays() method 179, 403  
addYears() method 179, 404  
after() method 179, 405  
AnyException exception 322  
APIs/Business Object Array function block 122  
    Add Element 122  
    Duplicate 122  
    Equals 122  
    Get Element At 122  
    Get Elements 122  
    Get Last Index 122  
    Is Business Object Array 122  
    Max attribute value 123  
    Min attribute value 123  
    Remove All Elements 123  
    Remove Element 123  
    Remove Element At 123  
    Set Element At 123  
    Size 123  
    Sum 123  
    Swap 123  
    To String 123  
APIs/Business Object function block 118  
    Copy 118  
    Duplicate 118  
    Equal Keys 118  
    Equals 118  
    Exists 118  
    Get Boolean 118  
    Get Business Object 119  
    Get Business Object Array 119  
    Get Business Object Type 119  
    Get Double 119

- APIs/Business Object function block *(continued)*
  - Get Float 119
  - Get Int 119
  - Get Long 119
  - Get Long Text 119
  - Get Object 119
  - Get String 119
  - Get Verb 120
  - Is Blank 120
  - Is Business Object 120
  - Is Key 120
  - Is Null 120
  - Is Required 120
  - Iterate Children 120
  - Key to String 120
  - New Business Object 120
  - Set Content 120
  - Set Default Attribute Values 120
  - Set Keys 121
  - Set Value 121
  - Set Value with Create 121
  - Set Verb 121
  - Set Verb with Create 121
  - Shallow Equals 121
  - To String 121
  - Valid Data 121
- APIs/Business Object/Array function block 121
  - GetBusObj At 121
  - New Business Object Array 121
  - Set BusObj At 122
  - Size 122
- APIs/Business Object/Constants function block 122
  - Verb Create 122
  - Verb Delete 122
  - Verb Retrieve 122
  - Verb Update 122
- APIs/Database Connection function block 123
  - Begin Transaction 123
  - Commit 124
  - Execute Prepared SQL 124
  - Execute Prepared SQL with Parameter 124
  - Execute SQL 124
  - Execute SQL with Parameter 124
  - Execute Stored Procedure 124
  - Get Database Connection 124
  - Get Database Connection with Transaction 124
  - Get Next Row 124
  - Get Update Count 124
  - Has More Rows 124
  - In Transaction 125
  - Is Active 125
  - Release 125
  - Roll Back 125
- APIs/Identity Relationship function block 125
  - Add My Children 125
  - Delete All My Children 125
  - Delete My Children 125
  - Foreign Key Cross-Reference 126
  - Foreign Key Lookup 126
  - Maintain Child Verb 126
  - Maintain Composite Relationship 126
  - Maintain Simple Identity Relationship 126
  - Update My Children 127
- APIs/Maps function block 127
  - Get Adapter Name 127
  - Get Calling Context 127
  - Get Original Request Business Object 127
- APIs/Maps/Constants function block 127
  - Calling Context ACCESS\_REQUEST 127
  - Calling Context ACCESS\_RESPONSE 127
  - Calling Context EVENT\_DELIVERY 127
  - Calling Context SERVICE\_CALL\_FAILURE 127
  - Calling Context SERVICE\_CALL\_REQUEST 128
  - Calling Context SERVICE\_CALL\_RESPONSE 128
- APIs/Maps/Exception function block 128
  - Raise Map Exception 128
  - Raise Map Exception 1 128
  - Raise Map Exception 2 128
  - Raise Map Exception 3 128
  - Raise Map Exception 4 128
  - Raise Map Exception 5 129
  - Raise Map RunTimeEntity Exception 129
- APIs/Participant function block 129
  - Get Boolean Data 129
  - Get Business Object Data 129
  - Get Double Data 129
  - Get Float Data 129
  - Get Instance ID 129
  - Get Int Data 129
  - Get Long Data 129
  - Get Participant Name 129
  - Get Relationship Name 130
  - Get String Data 130
  - New Participant 130
  - New Participant in Relationship 130
  - Set Data 130
  - Set Instance ID 130
  - Set Participant Definition 130
  - Set Relationship Definition 130
- APIs/Participant/Array function block 131
  - Get Participant At 131
  - New Participant Array 131
  - Set Participant At 131
  - Size 131
- APIs/Participant/Constants function block 131
- APIs/Participant/Constants function block, Participant INVALID\_INSTANCE\_ID 131
- APIs/Relationship function block 131, 257
  - Add Participant 131
  - Add Participant Data 131
  - Add Participant Data to New Relationship 131
  - Create Relationship 132
  - Create Relationship with Participant 132
  - Deactivate Participant 132
  - Deactivate Participant By Data 132
  - Deactivate Participant By Instance 132
  - Deactivate Participant By Instance Data 132
  - Delete Participant 132
  - Delete Participant By Instance 132
  - Delete Participant By Instance Data 132
  - Delete Participant with Data 133
  - Get Next Instance ID 133
  - Retrieve Instances 133
  - Retrieve Instances for Participant 133
  - Retrieve Participants 133
  - Retrieve Participants with ID 133
  - Update Participant 133
  - Update Participant By Instance 133
  - Update Participant By Instance Data 133
- Application-specific business objects 3
- AppSpecificInfo attribute property 507
- Attribute
  - addressing in transformations 169
  - advanced settings 251

- Attribute (*continued*)
    - application-specific information 507
    - checking for key 336
    - column name 251
    - comments for 18, 35, 52, 62, 75
    - creating synonyms for automatic mapping 71
    - data type 18, 35, 345, 507
    - dependencies of 77
    - destination 5, 18
    - finding 37, 51, 72
    - joining 39
    - mapping automatically 65
    - maximum length 507
    - name 18, 35, 507
    - properties 507, 509
    - relationship 48, 52, 190, 257, 258
    - required 338, 507
    - source 18
    - specifying 328
    - splitting 41
    - unlinked 20, 51, 64, 72
    - validating 56, 84
  - Attribute value
    - adding together 358
    - blank 336
    - copying 38, 52
    - default 37, 88, 341, 507
    - null 297, 337
    - retrieving 333
    - retrieving as string 344
    - retrieving maximum 350, 351, 352
    - retrieving minimum 353, 354, 355
    - setting 339, 343
    - setting default value for 341
    - validating 187
    - validating data type 345
    - zero-length string 336
  - AttributeException exception 322
  - Automatic mapping
    - adding prefix or suffix 65
    - attribute 65
    - business objects 65
    - creating maps 65
    - creating synonyms for 71
    - example of 67
    - Reverse Map 68
    - setting preferences 65
    - using synonyms, example 72
- B**
- BaseCollaboration class
    - method summary 453
  - BaseDLM class 7, 11, 315, 325
    - defined 315
    - getDBConnection() 315
    - getName() 317
    - getRelConnection() 318
    - implicitDBTransactionBracketing() 319
    - isTraceEnabled() 319
    - logError() 320
    - logInfo() 320
    - logWarning() 320
    - method summary 315
    - releaseRelConnection() 323
    - trace() 324
  - before() method 179, 406
  - beginTran() method (deprecated) 303, 381
  - beginTransaction() method 217, 365
  - BiDiBOTransformation() 361
  - BiDiBusObjTransformation() 362
  - Bidirectional
    - languages 162
  - Bidirectional functionality
    - Activity Editor 162
  - bidirectional languages
    - Designing maps for 60
  - Bidirectionality
    - Web services 162
  - BiDiStringTransformation() 363
  - Blank attribute value 336
  - BOOL\_TYPE constant 390
  - Boolean class 507
    - as stored-procedure parameter type 216, 308, 377
    - converting to 392
    - converting to Boolean 395
    - determining data type 390
    - valid conversions 392
  - boolean data type
    - as stored-procedure parameter type 216, 377
    - checking for valid data 345
    - converting to 395
    - converting to Boolean 392
    - determining data type 390
    - getting attribute value 333
    - setting attribute to 339
    - valid conversions 392
  - Breakpoints 90, 93
  - Browsing a Project 239
  - Business object
    - adding 16, 239
    - adding to an array 348
    - addressing in transformations 169
    - business object definition for 335
    - comparing attribute values 331, 332
    - comparing key attribute values 330
    - copying 329
    - deleting 16, 75, 239
    - duplicating 330
    - generic 3, 180, 192
    - instance name 35, 169
    - key attribute in 336
    - mapping automatically 65
    - null attribute in 337
    - number in a business object array 358
    - properties 170
    - refreshing list of 19
    - removing from business object array 356, 357
    - required attribute in 338
    - retrieving attribute value 333, 344
    - retrieving from business object array 349
    - retrieving key attribute value 338
    - retrieving verb 336
    - setting attribute value 339, 340, 343
    - setting key values 341
    - setting value of 357
    - swapping in an array 358
    - temporary 171
    - transversing hierarchical 328
    - validating attribute data type 345
    - variable for 169
  - Business object array
    - adding attribute values together 358
    - adding business object to 348

- Business object array (*continued*)
  - comparing with another 349
  - duplicating 348
  - index 84, 89, 169
  - removing all elements from 356
  - removing element from 356, 357
  - retrieving a business object from 349
  - retrieving contents of 350
  - retrieving last index of 350
  - retrieving maximum attribute value from 350, 351, 352
  - retrieving minimum attribute value from 353, 354, 355
  - retrieving size of 358
  - retrieving values as string 359
  - reversing position of elements in 358
  - setting element of 357
- Business object definition, retrieving name of 335
- BusObj class 10, 327, 346
  - copy() 329
  - defined 327
  - deprecated methods 345
  - duplicate() 330
  - equalKeys() 330
  - equals() 331
  - equalsShallow() 332
  - exists() 332
  - getCount() 345
  - getKeys() 345
  - getLocale() 335, 342
  - getType() 335
  - getValues() 345
  - getVerb() 336
  - isBlank() 336
  - isKey() 336
  - isNull() 337
  - isRequired() 338
  - keysToString() 338
  - method summary 327
  - not() 345
  - set() 339, 346
  - setContent() 340
  - setDefaultAttrValues() 341
  - setKeys() 341
  - setVerb() 342
  - setVerbWithCreate() 343
  - setWithCreate() 343
  - toString() 344
  - validData() 345
- BusObjArray class 10, 347, 359
  - addElement() 348
  - defined 347
  - duplicate() 348
  - elementAt() 349
  - equals() 349
  - getElements() 350
  - getLastIndex() 350
  - max() 350
  - maxBusObjArray() 351
  - maxBusObjs() 352
  - method summary 347
  - min() 353
  - minBusObjArray() 354
  - minBusObjs() 355
  - removeAllElements() 356
  - removeElement() 356
  - removeElementAt() 357
  - setElementAt() 357
  - size() 358

- BusObjArray class (*continued*)
  - sum() 358
  - swap() 358
  - toString() 359

## C

- calcDays() method 179, 406
- calcWeekdays() method 179, 407
- CALL statement 211, 212, 304, 368, 369, 383
- Call-triggered flow 192
- Calling contexts 190
  - ACCESS\_REQUEST 190, 265
  - ACCESS\_RESPONSE 190, 265
  - checking value of 176
  - custom relationship and 295
  - EVENT\_DELIVERY 190, 265
  - example of 193
  - identity relationship and 265
  - logic based on 176
  - retrieving 457
  - SERVICE\_CALL\_FAILURE 191, 265
  - SERVICE\_CALL\_REQUEST 190, 265
  - SERVICE\_CALL\_RESPONSE 191, 265
  - setting 460
  - strInitiator 176
  - testing with 95
- CANNOTCONVERT constant 391
- Cardinality attribute property 507
- catch statement 102, 197, 297
- Child business objects
  - adding to parent/child relationship 437, 450
  - addressing 171
  - attribute comment for 52
  - cardinality of 246, 507
  - customizing for relationships 277
  - example of customizing for relationships 277
  - identity relationships 246
  - mapping 193
  - multiple-cardinality 44, 171, 194
  - removing from parent/child relationship 439, 450
  - setting verb for 444
  - single-cardinality 171, 193, 194
  - submaps for 44, 45
  - testing 89
  - verb 287
  - version number 507
- CLASSPATH environment variable 167, 169
- Classpath preference 167
- Collaboration API
  - CxExecution 453
- CollaborationException class 328
- CollabUtils.jar file 167
- commit() method (CwDBConnection) 217, 366
- commit() method (DtpConnection) 303, 323, 382
- Comparing
  - business object arrays 349
  - business object attribute values 331, 332
  - key attribute values 330
- Composite identity relationship 227, 229, 244, 274, 284
  - customizing map rules for 277
  - defining 245, 246, 274
  - main map 277
  - maintainChildVerb() and 278, 290
  - maintainCompositeRelationship() and 275, 446
  - managing child instances 282
  - participant type for 274

- Composite identity relationship (*continued*)
  - submap 281
- Connection
  - determining if active 219, 372
  - obtaining 204, 315
  - opening 204, 300
  - releasing 219, 373
  - transaction programming model 216, 315, 316
- Connection links, adding to function blocks 147, 152
- Connection pool 204, 219, 316, 373
- Connector
  - initiating mapping request 87, 191, 265
  - retrieving name of 457
  - setting name of 460
- ContainedObjectVersion attribute property 507
- Content-based logic 174
- Context menu (Activity Editor)
  - accessing 110
  - Add Comment 110
  - Add Description 110
  - Add Label 110
  - Add To do 110
  - Add To My Collection 110
  - Check for Unmatched Delimiters 83
  - Expression Builder 182, 200
  - Goto Line 84
  - New Constant 110
- Context menu (business object browser)
  - Copy 35
  - Refresh All 19
- Context menu (business object pane)
  - Add Business Object 34
  - Delete Business Object 75
- Context menu (business object window)
  - Delete 36
  - Properties 170
- Context menu (dest. data, attribute)
  - Clear Breakpoint 92
  - Set Breakpoint 90
- Context menu (dest. data, main object)
  - Collapse 91
  - Save To 94
- Context menu (Map Designer), accessing 27
- Context menu (map workspace)
  - Add Business Object 34
  - Delete 75
  - Map Properties 58
  - Paste As Input Object 35
  - Paste As Output Object 35
- Context menu (Relationship Designer)
  - accessing 242
  - Change Index 246
- Context menu (source data, child object)
  - Add Instance 88, 89, 90
  - Remove All Instances 90
  - Remove Instance 90
- Context menu (source data, main object)
  - Load From 90
  - Reset 88
  - Save To 89
- Context menu (Transformations)
  - Open 28
  - Open in New Window 28
  - View Source 28
- copy() method 329, 346
- Copying
  - attributes 38, 52

- Copying (*continued*)
  - business object 329
  - participant definitions 248
  - relationship definitions 248
- Create verb
  - conditionally set 285
  - foreignKeyXref() and 292, 293
  - maintainChildVerb() and 288, 289
  - maintainCompositeRelationship() and 276
  - maintainSimpleIdentityRelationship() and 265, 267, 269, 272
  - relationship instance 296
- create() method 187, 296, 464, 468, 474
- Cross-Reference transformation 18, 23, 37, 47, 84, 104
  - and ACCESS\_REQUEST calling context 265
  - and ACCESS\_RESPONSE calling context 272
  - and EVENT\_DELIVERY calling context 265
  - and SERVICE\_CALL\_FAILURE calling context 272
  - and SERVICE\_CALL\_REQUEST calling context 267
  - and SERVICE\_CALL\_RESPONSE calling context 269
  - behavior with calling contexts 265
  - defining for relationships 263
  - validating 56
- Custom transformation 18, 23, 37, 47, 48, 53, 104, 174, 257
- CwBidiEngine method summary 361
- CwDBConnection class 11, 365, 375
  - beginTransaction() 365
  - commit() 366
  - creating object of 204, 315
  - executePreparedSQL() 367
  - executeSQL() 368
  - executeStoredProcedure() 370
  - getUpdateCount() 371
  - hasMoreRows() 371
  - inTransaction() 372
  - isActive() 372
  - method summary 365
  - methods for calling stored procedures 211
  - methods for row access 206
  - methods for transaction management 217
  - nextRow() 373
  - release() 373
  - rollback() 374
- CwDBStoredProcedureParam class 11, 213, 377, 379
  - constructor 377
  - getParamType() 378
  - getValue() 379
  - method summary 377
- CwDBStoredProcedureParam() constructor 213, 377
- CwDBTransactionException exception 217, 219, 316, 366, 367, 374, 375
- cwExecCtx variable 190, 425, 438, 439, 441, 442, 447, 449, 451
- CWMapMessages.txt message file 178, 498
- CWMAPTYPE constant 425
- CxExecutionContext class 453
  - CxExecutionContext() 453
  - defined 453
  - getContext() 454
  - MAPCONTEXT 453
  - setContext() 454
- CxExecutionContext() constructor 453
- CxMissingIDException exception 499

## D

- Data conversion 39, 389
  - class for 389



- Data conversion (*continued*)
  - Java.lang methods 389
  - to boolean data type 395
  - to Boolean object 392
  - to double data type 395
  - to Double object 393
  - to float data type 396
  - to Float object 393
  - to int data type 397
  - to Integer object 394
  - to String object 398
  - valid conversions 392
- Data Transformation Package 10
- Data type
  - attribute 507
  - determining 389
  - determining if conversion is possible 390
- Data validation 185, 187
- Database
  - connecting to 204, 219, 315
  - executing a query in 204, 368, 369, 370
  - handling data from 205
  - modifying 208, 209
  - querying 205, 209, 371, 373
  - rows affected by last write 371
- DataValidationLevel map property 101, 189
- Date class 216, 377, 390, 392, 507
- Date formatting
  - adding days to date 402
  - adding weekdays to date 403
  - adding years to date 404
  - calculating days between dates 406
  - calculating weekdays between dates 407
  - comparing dates 405, 406
  - current date 181, 182, 401
  - example of 179
  - generic format 180, 409
  - getting day of the month 409, 410
  - getting day of the week 410, 411
  - getting earliest date from a list 415, 417
  - getting hour value 410
  - getting in specified or default format 422
  - getting milliseconds between 1/1/70 and date 413
  - getting minutes value 411, 418
  - getting month name 418, 419
  - getting month value 412, 418
  - getting most recent date from a list 413, 414
  - getting seconds value 412, 419
  - getting year 412, 420
  - parsing date according to format 401
  - reformatting to CrossWorlds date format 409
  - using full names of months 408, 420, 421
  - using short names of months 408, 421
  - using weekday names 408, 422
- Date function block 134
  - Add Day 134
  - Add Month 134
  - Add Year 134
  - Date After 134
  - Date Before 134
  - Date Equals 134
  - example of using 150
  - Format Change 134, 150
  - Get Day 135
  - Get Month 135
  - Get Year 135
  - Get Year Month Day 135
- Date function block (*continued*)
  - Now 135
- DATE\_TYPE constant 390
- Date/Formats function block 135
  - yyyy-MM-dd 135
  - yyyyMMdd 135
  - yyyyMMdd HHmmss 135
- deactivateParticipant() method 296, 475
- deactivateParticipantByInstance() method 297, 476
- Debug menu (Map Designer) 27
  - Advanced 27
  - Attach 27, 94
  - Breakpoints 27, 91
  - Clear All Breakpoints 27, 92
  - Continue 27, 93
  - Detach 27, 94
  - Run Test 27, 92
  - Step Over 27, 93
  - Stop Test Run 27
  - Toggle Breakpoint 27, 91
- Default attribute value 37, 341, 507
- DefaultValue attribute property 507
- DELETE statement 208, 302, 368, 369
- Delete verb
  - foreignKeyXref() and 293
  - maintainChildVerb() and 288, 289
  - maintainCompositeRelationship() and 276
  - maintainSimpleIdentityRelationship() and 266, 268, 269, 272
  - relationship instance 297
- deleteMyChildren() method 439
- deleteParticipant() method 296, 477
- deleteParticipantByInstance() method 296, 478
- Deprecated methods
  - BusObj class 345, 462
  - DtpConnection class 299, 381
  - Relationship class 485
  - UserStoredProcedureParam class 487
- Design mode (Activity Editor) 105
- Designer toolbar (Map Designer) 25
  - Add Business Object 34
  - All Attributes 24
  - Clear All Breakpoints 92
  - Compile 85
  - Continue 93
  - displaying 24, 26
  - Linked Attributes 24
  - Run Test 93
  - Step Over 93
  - Toggle Breakpoint 91
  - Unlinked Attributes 24
  - Validate 84
- Designing maps for bidirectional languages 60
- Destination business object 3, 5, 15, 185
  - adding to map 32, 34, 35
  - business object window 35
  - displaying 11, 19, 26, 60
  - execution order 17, 56, 77, 84
  - relationship and 223
  - setting verb of 36
  - variable for 169
  - verb 36, 285
- Diagram tab (Map Designer) 19
  - adding business object 34
  - business object browser 19, 24, 26
  - business object variables 169
  - business object window 19, 26, 35, 169



Diagram tab (Map Designer) *(continued)*

- calling a submap 45
- custom transformation 48
- default display 24
- deleting a transformation 75
- displaying attributes 37
- joining attributes 40
- key mappings 23
- map workspace 19, 173
- moving attribute 39
- setting attribute value 37
- splitting attribute 41
- temporary business object 173

Double class 507

- as stored-procedure parameter type 215, 308, 377
- converting to 393
- converting to Double 396
- converting to Float 394, 396
- converting to Integer 394, 397
- converting to String 398
- determining data type 390
- obtaining maximum value 351, 352
- obtaining minimum value 353, 354, 355
- valid conversions 392

double data type

- as stored-procedure parameter type 215, 377
- checking for valid data 345
- converting to 395
- converting to Double 393
- converting to Float 394, 396
- converting to Integer 394, 397
- converting to String 398
- determining data type 390
- getting attribute value 333
- setting attribute to 339
- valid conversions 392

DOUBLE\_TYPE constant 390

DtpConnection class (deprecated) 10, 11, 381, 387

- beginTran() 381
- commit() 382
- creating object of 318
- execStoredProcedure() 384
- executeSQL() 383
- getUpdateCount() 385
- hasMoreRows() 385
- inTransaction() 386
- method summary 381
- methods for calling stored procedures 304
- methods for row access 301
- methods for transaction management 303
- nextRow() 386
- rollback() 387

DtpDataConversion class 10, 389, 398

- CANNOT\_CONVERT 391
- defined 389
- getType() 389
- isOKToConvert() 390
- method summary 389
- OKTO\_CONVERT 391
- POTENTIAL\_DATA\_LOSS 391
- toBoolean() 392
- toDouble() 393
- toFloat() 393
- toInteger() 394
- toPrimitiveBoolean() 395
- toPrimitiveDouble() 395
- toPrimitiveFloat() 396

DtpDataConversion class *(continued)*

- toPrimitiveInt() 397
- toString() 398

DtpDate class 10, 399, 423

- addDays() 402
- addWeekdays() 403
- addYears() 404
- after() 405
- before() 406
- calcDays() 406
- calcWeekdays() 407
- DtpDate() 401
- get12MonthNames() 408
- get12ShortMonthNames() 408
- get7DayNames() 408
- getCWDate() 409
- getDayOfMonth() 409
- getDayOfWeek() 410
- getHours() 410
- getIntDay() 410
- getIntDayOfWeek() 411
- getIntMilliSeconds() 411
- getIntMinutes() 411
- getIntMonth() 412
- getIntSeconds() 412
- getIntYear() 412
- getMaxDate() 413
- getMaxDateBO() 414
- getMinDate() 415
- getMinDateBO() 417
- getMinutes() 418
- getMonth() 418
- getMSSince1970() 413
- getNumericMonth() 418
- getSeconds() 419
- getShortMonth() 419
- getYear() 420
- method summary 399
- rules for 399
- set12MonthNames() 420
- set12MonthNamesToDefault() 421
- set12ShortMonthNames() 421
- set12ShortMonthNamesToDefault() 421
- set7DayNames() 422
- set7DayNamesToDefault() 422
- toString() 422

DtpDate() constructor 179, 181, 401

DtpDateException exception 179

DtpMapService class 10, 425, 426

- method summary 425
- runMap() 425

DtpSplitString class 10, 427, 432

- defined 427
- DtpSplitString() 427
- elementAt() 428
- firstElement() 428
- getElementCount() 429
- getEnumeration() 430
- lastElement() 430
- method summary 427
- nextElement() 430
- prevElement() 431
- reset() 432

DtpSplitString() constructor 427

DtpUtils class 10, 433, 435

- method summary 433
- padLeft() 433

- DtpUtils class (*continued*)
  - padRight() 433
  - stringReplace() 434
  - truncate() 435
- duplicate() method 330, 348
- Duplicating
  - business object 330
  - business object array 348
- Dynamic relationship 253

## E

- Edit menu (Activity Editor) 109
  - Copy 109
  - Cut 109
  - Delete 109
  - Find 109
  - Goto Line 84, 109
  - Paste 109
  - Redo 109
  - Replace 109
  - Select All 109
  - Undo 109
- Edit menu (Map Designer) 26
  - Add Business Object 26, 34, 170, 171
  - Delete Business Object 26, 75
  - Delete Current Selection 26, 36, 75
  - Find 26, 37, 51, 72
  - Insert Row 26
  - Map Properties 26, 58, 101, 167, 170, 189
  - Replace 26, 73
  - Select All 26
- Edit menu (Relationship Designer) 241
  - Advanced Settings 242, 245, 247, 249, 252
  - Copy 242, 248
  - Cut 242
  - Delete 253
  - Paste 242, 248
  - Rename 242
- elementAt() method 349, 428
- Enumeration class 206, 301
- Environment variable
  - CLASSPATH 167, 169
  - JCLASSES 167
  - PATH 11
- equalKeys() method 330
- equals() method 331, 349
- equalsShallow() method 332
- Error
  - compilation 83, 86
  - run-time 100
- Error message 320, 503
- EVENT\_DELIVERY calling context 190, 191, 265
  - Create verb and 265, 288, 292, 296
  - Delete verb and 266, 288, 297
  - foreignKeyXref() and 291, 443
  - getOriginalRequestBO() and 459
  - maintainChildVerb() and 288, 446
  - maintainCompositeRelationship() and 276, 447
  - maintainSimpleIdentityRelationship() and 265, 450
  - original-request business object 192, 459
  - Retrieve verb and 266, 288, 292
  - retrieving 458
  - setting to 461
  - testing with 96, 99
  - Update verb and 266, 288, 292
  - updateMyChildren() and 284
- Event-triggered flow 190, 191
- Exception handling 186, 187
- Exception types 328
- Exceptions
  - CollaborationException class 328
  - CwDBTransactionException 217, 219, 316, 366, 367, 374, 375
  - defined 186, 328
  - raising 321, 502
  - RelationshipRuntimeException class 186
  - relationships 186
  - type 328
- execStoredProcedure() method (deprecated) 304, 305, 384
- executePreparedSQL() method 209, 211, 212, 367
- executeSQL() method (CwDBConnection) 205, 211, 212, 368
- executeSQL() method (DtpConnection) 300, 304, 383
- executeStoredProcedure() method 211, 212, 370
- execution context 453
- exists() method 332
- Explicit transaction bracketing 216
  - releasing the connection 219
  - scope of transaction 217
- Expression Builder 182, 200

## F

- File menu (Activity Editor) 108
  - Close 108
  - Print 108
  - Print Preview 108
  - Print Setup 108
  - Save 108, 176, 183
- File menu (Map Designer) 25
  - Close 25, 57
  - Compile 26, 57, 85, 88, 195, 199, 202
  - Compile All 26, 85
  - Compile with Submap(s) 26, 85
  - Create Map Document 26, 63
  - Delete 26, 76
  - Exit 26, 57
  - New 25, 31
  - Open 25, 56, 57
  - Print 26, 74
  - Print Preview 26, 74
  - Print Setup 26, 74
  - Save 25, 49, 51
  - Save As 26, 49, 51
  - Validate Map 26, 84
  - View Map Document 26, 64
- File menu (Relationship Designer) 241
  - Add Participant Definition 241, 243
  - New 241
  - New Relationship Definition 243
  - Save 241
  - Save All 241
  - Save Relationship Definition 243, 248
  - Switch to Project 241
- Find and Replace text 73
- Find text 72
- Find unlinked attribute 72
- firstElement() method 428
- Float class 507
  - as stored-procedure parameter type 215, 308, 377
  - converting to 393
  - converting to Double 393, 396
  - converting to Float 396
  - converting to Integer 394, 397

- Float class (*continued*)
  - converting to String 398
  - determining data type 390
  - obtaining maximum value 351, 352
  - obtaining minimum value 353, 354, 355
  - valid conversions 392
- float data type
  - as stored-procedure parameter type 215, 377
  - checking for valid data 345
  - converting to 396
  - converting to Double 393, 396
  - converting to Float 394
  - converting to Integer 394, 397
  - converting to String 398
  - determining data type 390
  - getting attribute value 333
  - setting attribute to 339
  - valid conversions 392
- FLOAT\_TYPE constant 390
- Foreign key 290, 440, 442, 507
- Foreign Key Cross-Reference function block 291, 294
- Foreign key lookup 52, 290
- Foreign Key Lookup function block 290, 294
- foreignKeyLookup() method 52, 290, 297, 298, 440, 499
- foreignKeyXref() method 53, 291, 297, 298, 299, 442, 499
- Function blocks 112, 116
  - adding connection links 146, 151
  - adding custom Jar libraries as 164
  - customizing Jar library properties 165
  - dragging and dropping 146, 150
  - example of using 144, 150, 157
  - General/APIs/Business Object 118
  - General/APIs/Business Object Array 122
  - General/APIs/Business Object/Array 121
  - General/APIs/Business Object/Constants 122
  - General/APIs/Database Connection 123
  - General/APIs/Identity Relationship 125
  - General/APIs/Maps 127
  - General/APIs/Maps/Constants 127
  - General/APIs/Maps/Exception 128
  - General/APIs/Participant 129
  - General/APIs/Participant/Array 131
  - General/APIs/Participant/Constants 131
  - General/APIs/Relationship 131
  - General/Date 134
  - General/Date/Formats 135
  - General/Logging and Tracing 135
  - General/Logging and Tracing/Log Error 136
  - General/Logging and Tracing/Log Information 136
  - General/Logging and Tracing/Log Warning 137
  - General/Logging and Tracing/Trace 137
  - General/Mapping 138
  - General/Math 138
  - General/Properties 140
  - General/Relationship 141
  - General/String 141
  - General/Utilities 143
  - General/Utilities/Vector 144
  - moving 146, 151
  - using directly in Map Designer 112
  - using to implement relationships 257
  - Web services 159

- get7DayNames() method 179, 408
- getConnName() method 457
- getContext() method 454
- getCount() method (deprecated) 345
- getCWDate() method 179, 180, 409
- getDayOfMonth() method 179, 409
- getDayOfWeek() method 179, 410
- getDBConnection() method 204, 216, 315, 316
- getElementCount() method 429
- getElements() method 350
- getEnumeration() method 430
- getGenericBO() method (deprecated) 462
- getHours() method 179, 410
- getInitiator() method 190, 457
- getInstanceId() method 187, 465
- getIntDay() method 179, 410
- getIntDayOfWeek() method 179, 411
- getIntMilliseconds() method 411
- getIntMinutes() method 179, 411
- getIntMonth() method 179, 412
- getIntSeconds() method 179, 412
- getIntYear() method 179, 412
- getKeys() method (deprecated) 345
- getLastIndex() method 350
- getLocale() method 335, 342, 458
- getMaxDate() method 179, 413
- getMaxDateBO() method 179, 414
- getMinDate() method 179, 415
- getMinDateBO() method 179, 417
- getMinutes() method 179, 418
- getMonth() method 179, 418
- getMSSince1970() method 179, 413
- getName() method 317
- getNewID() method 479
- getNumericMonth() method 179, 418
- getOriginalRequestBO() method 190, 459, 462
- getParamDataJavaObj() method (deprecated) 305, 308, 488
- getParamDataJDBC() method (deprecated) 305, 308, 489
- getParamIndex() method (deprecated) 305, 489
- getParamIOType() method (deprecated) 305, 490
- getParamName() method (deprecated) 305, 491
- getParamType() method 213, 378
- getParamValue() method (deprecated) 305, 491
- getParticipantDefinition() method 466
- getRelationshipDefinition() method 466
- getRelConnection() method (deprecated) 300, 318, 383, 384
- getSeconds() method 179, 419
- getShortMonth() method 179, 419
- getType() method 335, 389
- getUpdateCount() method (CwDBConnection) 208, 371
- getUpdateCount() method (DtpConnection) 303, 385
- getValue() method 213, 379
- getValues() method (deprecated) 345
- getVerb() method 336
- getYear() method 179, 420
- Graphical view (Activity Editor) 104, 105
  - Content window 106
  - Design mode 105
  - Library window 105
  - Properties window 106
  - Quick view mode 106
- Graphics toolbar (Activity Editor) 111
  - Back 111
  - Forward 111
  - Home 111
  - Up One Level 111

## G

- get12MonthNames() method 179, 408
- get12ShortMonthNames() method 179, 408

Graphics toolbar (Activity Editor) *(continued)*

- Zoom In 111
- Zoom Out 111

## H

- hasMoreRows() method (CwDBConnection) 206, 211, 371
- hasMoreRows() method (DtpConnection) 301, 385
- Help menu (Activity Editor) 110
  - documentation 110
  - Help Topics 110
- Help menu (Map Designer) 27
  - About Map Designer 27
  - Documentation 27
  - Help Topics 27
- Help menu (Relationship Designer) 242
  - About Relationship Designer 242
  - Documentation 242
  - Help topics 242
- Hierarchical business object
  - comparing all 331
  - comparing top-level 332
  - transversing 328

## I

- Identity relationship 225, 229
  - adding child business objects 437, 450
  - child business objects 246
  - class for 437
  - creating participant for 465
  - defined 224, 225, 244
  - defining 244, 246, 263, 273, 274
  - deleting child business objects 439, 450
  - kinds of 225, 244
  - maintaining child verb 287
  - relationship instance IDs 232
  - static 254
  - static lookup 285
  - testing 95
- IdentityRelationship class 10, 231, 437, 452
  - addMyChildren() 437, 485
  - deleteMyChildren() 439
  - foreignKeyLookup() 440
  - foreignKeyXref() 442
  - maintainChildVerb() 444
  - maintainCompositeRelationship() 446, 485
  - maintainSimpleIdentityRelationship() 448, 485
  - method summary 437
  - updateMyChildren() 450, 485
- Implementing relationships using function blocks 257
  - Composite identity relationship 275
  - Foreign Key Cross-Reference 291, 294
  - Foreign Key Lookup 290, 294
  - Identity relationship 273
  - Maintain Child Verb 273, 276, 278, 288, 289
  - Update My Children 279, 284
- Implicit transaction bracketing 216
  - as default 216
  - releasing the connection 219
  - scope of transaction 217
- implicitDBTransactionBracketing() method 217, 319
- import statement 167
- IN parameter 213, 214, 379
- Inbound map 3, 4
  - example of customizing 285

Inbound map *(continued)*

- foreign key lookup in 293, 441, 443
  - in map document 62
  - lookup relationship in 261, 262, 481
  - testing 96, 97, 99
- Informational message 320, 503, 504
  - INOUT parameter 213, 379
  - INSERT statement 208, 260, 302, 368, 369, 371
  - int data type
    - as stored-procedure parameter type 215, 377
    - checking for valid data 345
    - converting to 397
    - converting to Double 393, 396
    - converting to Float 394, 396
    - converting to Integer 394
    - converting to String 398
    - determining data type 390
    - getting attribute value 333
    - setting attribute to 339
    - valid conversions 392
  - Integer class 507
    - as stored-procedure parameter type 215, 308, 377
    - converting to 394
    - converting to Double 393, 396
    - converting to Float 394, 396
    - converting to Integer 397
    - converting to String 398
    - determining data type 390
    - obtaining maximum value 351, 352
    - obtaining minimum value 353, 354, 355
    - valid conversions 392
  - INTEGER\_TYPE constant 390
  - inTransaction() method (CwDBConnection) 217, 219, 372
  - inTransaction() method (DtpConnection) 303, 386
  - INVALID\_INSTANCE\_ID constant 466, 468, 474
  - isActive() method 219, 372
  - isBlank() method 336
  - IsForeignKey attribute property 507
  - IsKey attribute property 507
  - isKey() method 336
  - isNull() method 337
  - isOKToConvert() method 390
  - IsRequired attribute property 507
  - isRequired() method 338
  - isTraceEnabled() method 319
- ## J
- Jar libraries
    - customizing display settings 165
    - importing as function blocks 164
  - Java class
    - Boolean 392, 507
    - Date 390, 507
    - Double 393, 396, 507
    - Enumeration 206, 301
    - Float 393, 396, 507
    - Integer 394, 397, 507
    - java.sql.Types 215, 307
    - Object 333, 339, 345
    - StringTokenizer 427
    - Vector 206, 213, 301, 368, 373, 378
  - Java compiler (javac) 85
  - Java Development Kit (JDK) 11, 166
  - Java operator, NOT 345
  - Java statements
    - catch 102, 297

- Java statements *(continued)*
  - import 167
  - try 101, 297
- Java toolbar (Activity Editor) 111
  - Edit Java Code 111
  - Expression Builder 111
  - Find Text 111
  - Goto Line 111
  - Redo 111
  - Undo 111
- Java view (Activity Editor) 104
  - Design mode 107
  - Quick view mode 107
  - WordPad 107
- java.lang package 389
- java.sql.Types class 215, 307
- java.util package 206, 301, 427
- JavaException exception 322
- JCLASSES environment variable 167
- Join transformation 18, 23, 36, 39, 52, 56, 84, 104

## K

- Key attribute 225, 507
  - composite 227, 245, 274, 447
  - foreign 290, 440, 442, 507
  - identity relationships and 245
  - single 225, 245, 449
- Key attribute values
  - checking for 336
  - comparing 330
  - retrieving as string 338
  - setting 341
- Keyboard shortcut 29
- keysToString() method 338, 345

## L

- languages
  - Designing maps for bidirectional 60
- lastElement() method 430
- logError() method 178, 320, 502, 503
- Logging 101, 178, 503, 504
  - example 503
  - levels 504
  - methods that send message 320, 502, 503
  - principles of 504
  - severity levels 503
- Logging and Tracing function block 135
  - Log error 135
  - Log error ID 135
  - Log information 135
  - Log information ID 135
  - Log warning 136
  - Log warning ID 136
  - Trace 136
- Logging and Tracing/Log Error function block 136
  - Log error ID 1 136
  - Log error ID 2 136
  - Log error ID 3 136
- Logging and Tracing/Log Information function block 136
  - Log information ID 1 136
  - Log information ID 2 136
  - Log information ID 3 137
- Logging and Tracing/Log Warning function block 137
  - Log warning ID 1 137

- Logging and Tracing/Log Warning function block *(continued)*
  - Log warning ID 2 137
  - Log warning ID 3 137
- Logging and Tracing/Trace function block 137
  - Trace ID 1 137
  - Trace ID 2 138
  - Trace ID 3 138
  - Trace on Level 138
- Logical operator 345
- logInfo() method 101, 102, 203, 320, 502, 503, 504
- logWarning() method 320, 503
- long data type 215, 333, 339, 345, 377
- LongText class
  - determining data type 390
  - getting attribute value 333
  - obtaining maximum value 351, 352
  - obtaining minimum value 353, 354, 355
  - setting attribute 340
  - valid conversions 392
- LONGTEXT\_TYPE constant 390
- Lookup relationship 224, 258, 262
  - code for 261, 481, 482
  - creating participant for 465
  - defined 224, 246, 258
  - defining 246, 259
  - example of 224, 259
  - participant type for 234, 247, 259
  - relationship instance IDs 232
  - static 157, 254
  - testing 98

## M

- Maintain Child Verb function block 276, 288, 289
- Maintain Composite Identity Relationship function block 275
- maintainChildVerb() method 276, 287, 290, 444
  - validations performed 445
- maintainCompositeRelationship() method 446
  - actions of 275
  - attribute comment for 52
  - deprecated version 485
  - error messages 298, 498, 499
- maintainSimpleIdentityRelationship() method 448
  - attribute comment for 52
  - deprecated version 485
  - error messages 297, 298, 498, 499
  - validations performed 449
- Managing child instances function blocks 282
- map 453
- Map automation
  - creating maps automatically 65
  - creating reverse maps 68
  - creating synonyms for 71
- Map definition 5, 7
  - creating 31
  - defined 5
  - in map definition file 50
  - loading 79
  - location of 5
  - naming conventions 5
  - New Map wizard 31
  - unloading 79
- Map Designer 8, 15, 55
  - Add Business Object dialog 34
  - Automatic mapping 65
  - Breakpoint dialog 92
  - business object browser 19, 24, 26



- Map Designer (*continued*)
  - business object pane 19, 24, 34, 75, 173
  - business object window 19, 26, 35, 169
  - Condition for calling Submap dialog 199
  - Context menu 27
  - data conversion by 39
  - Debug menu 27
  - Delete Business Object dialog 75
  - Delete Map dialog 76
  - Edit menu 26
  - exiting 26, 57
  - File menu 25
  - files generated 13
  - Find control pane 25, 51, 72
  - functionality of 25
  - Help menu 27
  - launching 15
  - layout of 16
  - main components 16
  - main window 16, 24
  - map workspace 19, 34, 173
  - menus of 25
  - Messages tab 20, 24, 497
  - Multiple Attributes dialog 18, 40
  - New Map wizard 31, 34
  - Open file with map dialog 57
  - Open Map from Project dialog 56
  - output window 17, 21, 24, 25, 26, 86, 87
  - overview 15
  - preferences 21
  - Programs toolbar 24, 25, 26
  - Reverse map 65
  - Save Map As dialog 34, 49
  - search facility 72
  - starting 15
  - status bar 17, 24, 27
  - Submap dialog 45, 195, 198, 199
  - tab window 8, 55
  - Tab window 17
  - Test tab 20, 24, 87
  - toolbars 24, 25, 28
  - Tools menu 27
  - View menu 26
  - working in projects 16
- Map development 11, 15
- Map document 60
- Map execution
  - continuing 93
  - execution order 17, 56, 77, 84
  - map instances and 7
  - pausing 90, 93
  - purpose of 190
  - relationship instances and 230, 234
  - test run and 87
  - transactions and 219, 309, 316, 318
  - viewing 87, 93
- Map execution context 189
  - calling context 190, 458, 459, 461, 462
  - class for 189, 457
  - cxExecCtx 190
  - original-request business object 192, 268, 269, 273, 292, 293, 460
- Map initiators xv
- Map instance 7
  - calling context 457, 460
  - class for 7, 315
  - connector name 457, 460
- Map instance (*continued*)
  - contents of 7
  - data validation level 189
  - defined 7
  - execution context 7, 189
  - original-request business object 459
  - reusing 172, 185
  - starting 189, 506
  - stopping 189, 506
  - trace level 506
  - transaction programming model 216, 319
- Map properties 11, 58
  - DataValidationLevel 101, 189
  - Map file declaration block 167
  - Map local declaration block 167
  - run-time 59
  - Trace level 324, 505
  - updating from server component management view 59, 186, 189
- Map Properties dialog (Map Designer)
  - Business Objects tab 169, 170, 173
  - General tab 59, 167, 173, 185, 189, 216, 316
- Map Properties dialog (Map Designers)
  - General tab 505
- Map repository file 79
- Map Utilities package 166, 167
- MAPCONTEXT constant 453
- MapExeContext class 11, 457, 463
  - calling-context constants 191
  - deprecated methods 462
  - getConnName() 457
  - getGenericBO() 462
  - getInitiator() 457
  - getLocale() 458
  - getOriginalRequestBO() 459
  - method summary 457
  - setConnName() 460
  - setInitiator() 460
  - setLocale() 461
- MapFailureException exception 297
- mapName.\_locale.txt message file 497
- mapName.txt message file 499
- Mapping
  - automatic 65
  - defined 3
  - overview 3
  - polymorphic 78
  - reverse 65
  - simple 5
  - standards 52, 101, 297
  - support for 3
  - tools for 8, 9
- Mapping API 10, 52
  - BaseDLM class 11
  - business object classes 10
  - BusObj class 10
  - BusObjArray class 10, 347
  - CwDBConnection class 11, 365
  - CwDBStoredProcedureParam class 11, 377
  - DTP classes 10
  - DtpConnection class 10, 11, 381
  - DtpDataConversion class 10, 389
  - DtpDate class 10, 399
  - DtpMapService class 10, 425
  - DtpSplitString class 10, 427
  - DtpUtils class 10, 433
  - IdentityRelationship class 10, 437

- Mapping API (*continued*)
  - MapExeContext class 11, 457
  - Participant class 10, 463
  - Relationship class 10, 471
  - relationship classes 10
  - UserStoredProcedureParam class 11, 487
  - utility classes 11
- Mapping function block 138
  - Run Map 138
  - Run Map with Context 138
- Mapping role 59
- Maps
  - base class for 315
  - closing 57
  - coding 103, 219
  - compiling 17, 20, 50, 84, 86, 88, 167
  - converting 80
  - creating 30
  - creating automatically 65
  - creating reverse maps 68
  - current 49, 55, 85, 173, 317
  - debugging 94, 100
  - defined 3, 8, 15
  - deleting 76
  - development files 13
  - exceptions and 186
  - execution context 189
  - HTML version 60
  - improving modularity of 43
  - map documents 60
  - name of 5, 33, 59, 317
  - naming 34
  - opening 55
  - polymorphic 78
  - printing 74
  - renaming 50
  - saving 20, 34, 49, 75
  - saving to file 51
  - saving to project 49
  - testing 87, 94
  - using Web services in 159
  - validating 22, 49, 55, 84
  - viewing execution 87, 93
  - working with 55
  - XML version 50
- maps for bidirectional languages
  - Designing 60
- Math function block 138
  - Absolute value 138
  - Ceiling 138
  - Divide 139
  - Equal 139
  - Floor 139
  - Greater than 139
  - Greater than or Equal 139
  - Less than 139
  - Less than or equal 139
  - Maximum 139
  - Minimum 139
  - Minus 140
  - Multiply 140
  - Not a Number 140
  - Not Equal 140
  - Number to String 140
  - Plus 140
  - Round 140
  - String to Number 140
- MAX\_CONNECTION\_POOLS configuration parameter 250, 252
- max() method 350
- maxBusObjArray() method 351
- maxBusObjs() method 352
- MaxLength attribute property 507
- Message 20
  - 5000 297, 498, 499
  - 5001 298, 498, 500
  - 5002 298, 499, 500
  - 5003 297, 499, 500
  - 5007 298, 499, 500
  - 5008 298, 499, 500
  - 5009 298, 299, 499, 500
  - format 500
  - location of 20, 497
  - number 501
  - parameters in 497, 501
  - revising 504
  - severity 503, 504
  - text 501
- Message file 497, 506
  - choosing which one to use 497
  - comments 502
  - CWMapMessages.txt 178, 498
  - defined 497
  - displaying 20
  - format 500
  - location of 13, 497
  - maintaining 502
  - mapName\_locale.txt 497
  - mapName.txt 499
  - operations that use 502
  - overview 497
  - UserMapMessages.txt 498, 499
  - using 178, 501, 503
- Message logging 365
- method summary, CwBidiEngine 361
- min() method 353
- minBusObjArray() method 354
- minBusObjs() method 355
- Move transformation 18, 23, 36, 38, 52, 56, 84
- Multiple-map map table 62

## N

- Name attribute property 507
- Naming conventions
  - maps 5
  - participant definitions 233, 243
  - relationship definitions 230, 243
- New Constant 110, 115, 153, 154
- nextElement() method 430
- nextRow() method (CwDBConnection) 206, 211, 373
- nextRow() method (DtpConnection) 301, 386
- Non-identity relationships 224
- NOT operator 345
- not() 345
- Null attribute value 297, 337
- Numbers, truncating 435

## O

- Object class 333, 339, 345
- ObjectEventId attribute 52, 84, 88, 95
- ObjectException exception 322

- OKTOCONVERT constant 391
- OperationException exception 322
- Original-request business object 192, 268, 269, 273, 292, 293, 459
- OUT parameter 212, 213, 214, 379
- Outbound map 3, 5
  - example of customizing 286
  - foreign key lookup in 441, 443
  - in map document 62
  - lookup relationship in 261, 262, 482
  - testing 96, 99

## P

- Package
  - Data Transformation 10
  - importing Java packages 164
  - java.lang 389
  - java.util 206, 301, 427
  - Map Utilities 166, 167
- padLeft() method 433
- padRight() method 433
- PARAM\_IN constant 214, 379
- PARAM\_INOUT constant 379
- PARAM\_OUT constant 214, 379
- Parent/child relationship 282
  - adding child instance 437, 450
  - defined 282
  - defining 246
  - deleting child instance 439, 450
- Participant class 10, 231, 235, 463, 469
  - defined 463
  - getInstanceId() 465
  - getParticipantDefinition() 466
  - getRelationshipDefinition() 466
  - method summary 463
  - Participant() 463
  - set() 467
  - setInstanceId() 467
  - setParticipantDefinition() 468
  - setRelationshipDefinition() 468
- Participant definition 233
  - advanced settings 245, 250
  - copying 248
  - creating 243
  - defined 233
  - location of 233
  - name of 466, 468
  - naming conventions 233, 243
  - renaming 249
- Participant instance 234
  - adding to relationship instance 472
  - class for 235, 463
  - constructor for 463
  - contents of 234
  - creating 296, 463, 474
  - data 235, 251, 463, 465, 467
  - deactivating 475, 476
  - defined 234, 463
  - deleting 296, 477, 478
  - identifier 234
  - participant definition 235, 463, 466, 468
  - relationship definition 234, 463, 466, 468
  - relationship instance ID 234, 463, 465, 467, 480
  - retrieving from relationship instance 482
  - updating 483, 484
- Participant instance identifier 234

- Participant type 233, 243
  - business object 234, 243, 244, 263, 274
  - Data 224, 234, 243, 247, 259
- Participant Types window 240, 241, 243, 247
- Participant() constructor 463, 468
- Participants 233, 235
  - defined 223
  - naming conventions 233, 243
- PATH environment variable 11, 85
- Polymorphic maps 78
- POTENTIALDATALOSS constant 391
- Preferences dialog (Map Designer) 27
  - Automatic Mapping tab 23, 67
  - Custom Mapping tab 23
  - General tab 17, 18, 22, 50, 55, 76, 77, 78, 85, 170
  - Key Mapping tab 23, 39, 40, 42, 45, 48
  - Validation tab 22
- prevElement() method 431
- Project 16, 238
  - browsing a 239
  - opening a map from 16
  - saving map to 49
  - saving the map in 16, 238
  - working in 16
  - working with 238
- Properties function block 140
- Properties function block, Get Property 140

## Q

- Quick view mode (Activity Editor) 106

## R

- Relationship attribute property 507
- Relationship class 10, 231, 471, 487
  - addParticipant() 472
  - create() 474
  - deactivateParticipant() 475
  - deactivateParticipantByInstance() 476
  - defined 471
  - deleteParticipant() 477
  - deleteParticipantByInstance() 478
  - deprecated methods 485
  - getNewID() 479
  - guidelines 471
  - method summary 471
  - retrieveInstances() 480
  - retrieveParticipants() 482
  - updateParticipant() 483
  - updateParticipantByInstance() 484
- Relationship database 230
  - connecting to 300, 309, 318
  - determining if transaction is in progress 386
  - disconnecting from 323
  - executing a query in 300
  - location of 13, 230, 231, 250, 251, 252
  - modifying 302
  - queries for more rows to process 385
  - retrieving next row 386
  - rows affected by last write 385
  - SQL queries 10, 383
  - type of 250, 251
  - user account for 249, 250, 251
- Relationship definition 229, 230
  - advanced settings 245, 249



- Relationship definition (*continued*)
  - changing 244
  - copying 248
  - creating 243
  - defined 8, 229, 237
  - deleting 253
  - dependent objects 311
  - identity 244, 246, 263, 273, 274
  - list 239
  - loading 310
  - location of 229
  - lookup 246, 259
  - name of 466, 468
  - naming conventions 230, 243
  - parent/child 282
  - renaming 249
  - saving 243
  - unloading 309
  - viewing 239
- Relationship Designer 8
  - Advanced Settings dialog 245, 249, 252, 253, 254
  - Edit menu 241
  - File menu 241
  - functionality of 241
  - Global Default Settings dialog 252
  - Help menu 242
  - launching 237
  - layout of 239
  - main window 240
  - menus of 241
  - overview 237
  - Programs toolbar 238
  - starting 237
  - status bar 240, 242
  - toolbar 242
  - toolbars 240
  - Tools menu 242
  - View menu 242
  - working with projects 238
- Relationship development 235
- Relationship function block 141, 257
  - example of using 158
  - Maintain Identity Relationship 141
  - Static Lookup 141, 158
- Relationship instance 230, 233
  - adding a participant to 472
  - class for 7, 231, 437, 471
  - creating 296, 474
  - creating participant for 296, 464
  - deactivating participant 296, 475, 476
  - defined 230
  - deleting child objects 439
  - deleting participant 296, 477, 478
  - location of 231
  - retrieving instance ID 479, 480
  - retrieving participants from 482
  - run-time data 248
  - updating participants 483, 484
- Relationship instance ID 232
  - deactivating participant by 477
  - defined 232
  - deleting participant by 479
  - duplicate 187
  - identity relationship and 232
  - in participant instance 465, 467
  - lookup relationship and 232
  - retrieving for participant 480
- Relationship instance ID (*continued*)
  - retrieving next 479
  - updating participant by 484
- Relationship Manager 261
- Relationship repository file 309
- Relationship tables 230, 231
  - caching 253
  - changes to 193
  - composite identity relationships 275
  - contents of 233
  - creating 245, 248, 251
  - defined 231
  - foreign 290, 441, 443
  - foreign key lookups and 440, 442
  - identity relationships 263
  - index size 275
  - location of 230, 231, 250, 251, 252, 253
  - lookup relationships 157, 246, 247, 259
  - MaxLength attribute 275
  - modifying 302, 442
  - name of 231, 251, 259
  - participants in 476, 477
  - performing lookup in 285
  - querying 300
  - table schemas 248, 249, 251
- RelationshipRuntimeException class 97, 186, 298, 498
- Relationships 229, 233
  - custom 295
  - defined 223
  - dynamic 253
  - exceptions 186
  - implementing code for 257
  - introduction 223, 236
  - naming conventions 230, 243
  - non-identity 224
  - optimizing 253
  - starting 244
  - static 253, 254
  - static lookup 157
  - stopping 244
  - testing 95
  - transformations for 48, 52, 257
  - types of 223, 249
  - working with 257
- release() method 219, 373
- releaseRelConnection() method (deprecated) 323
- removeAllElements() method 356
- removeElement() method 356
- removeElementAt() method 357
- Replace text 73
- repos\_copy utility 79, 309
- Repository
  - exporting a map from 79
  - exporting a relationship 309
  - location of relationship tables 300
  - relationship database and 230, 231
- Required attribute 338, 507
- reset() method 432
- Retrieve verb 447, 449
  - foreignKeyXref() and 292, 293
  - maintainChildVerb() and 288, 289
  - maintainCompositeRelationship() and 276
  - maintainSimpleIdentityRelationship() and 266, 268, 269, 272
- retrieveInstances() method 52, 262, 285, 480
- retrieveParticipants() method 52, 262, 285, 482

- retrieving
  - configuration property value 453, 454
- Retrieving
  - business object array contents 350
  - business object array maximum value 350, 351, 352
  - business object array minimum value 353, 354, 355
  - business object array values as string 359
  - business object attribute 333
  - business object from array 349
  - business object key attribute value as string 338
  - business object type 335
  - business object verb 336
  - last index from business object array 350
  - map name 317
  - number of elements in business object array 358
- Reverse map
  - creating automatically 68, 69
  - example of 69
  - providing delimiters 69
  - transformation rules 68
- rollBack() method (CwDBConnection) 217, 374
- rollBack() method (DtpConnection) 303, 387
- runMap() method 46, 78, 190, 195, 198, 200, 202, 295, 425

## S

- SELECT statement 205, 209, 300, 368, 369, 372, 373
- Server component management view
  - updating map properties 59, 186, 189
  - updating relationship properties 255
- SERVICE\_CALL\_FAILURE calling context 191, 265
  - generic business object and 192
  - getOriginalRequestBO() and 459
  - maintainCompositeRelationship() and 276
  - maintainSimpleIdentityRelationship() and 272
  - original-request business object 192, 459
  - retrieving 458
  - setting to 461
- SERVICE\_CALL\_REQUEST calling context 190, 191, 265
  - Create verb and 267, 288, 293
  - Delete verb and 268, 288, 293
  - foreignKeyXref() and 293, 443
  - generic business object and 192
  - getOriginalRequestBO() and 459
  - maintainChildVerb() and 288, 446
  - maintainCompositeRelationship() and 276, 447
  - maintainSimpleIdentityRelationship() and 267, 449, 450
  - original-request business object 192, 459
  - Retrieve verb and 268, 288, 293
  - retrieving 458
  - setting to 461
  - testing with 97, 99
  - Update verb and 268, 288, 293
  - updateMyChildren() and 284
- SERVICE\_CALL\_RESPONSE calling context 191, 265
  - Create verb and 269, 289, 292, 296
  - Delete verb and 269, 289, 297
  - foreignKeyXref() and 291, 443
  - generic business object and 97, 192
  - getOriginalRequestBO() and 459
  - identity relationships and 97
  - maintainChildVerb() and 288, 446
  - maintainCompositeRelationship() and 276, 447
  - maintainSimpleIdentityRelationship() and 269, 450
  - original-request business object 192, 271, 459
  - Retrieve verb and 269, 289, 292
  - retrieving 458

- SERVICE\_CALL\_RESPONSE calling context (*continued*)
  - setting to 461
  - testing with 98, 100
  - Update verb and 269, 289, 292
  - updateMyChildren() and 284
- ServiceCallException exception 322
- Set Value transformation 18, 36, 37, 52, 56, 84, 104
- set() method 339, 346, 467
- set12MonthNames() method 179, 420
- set12MonthNamesToDefault() method 179, 421
- set12ShortMonthNames() method 179, 421
- set12ShortMonthNamesToDefault() method 179, 421
- set7DayNames() method 179, 422
- set7DayNamesToDefault() method 179, 422
- setConnName() method 460
- setContent() method 340
- setContext() method 454
- setDefaultAttrValues() method 341
- setElementAt() method 357
- setInitiator() method 190, 460
- setInstanceId() method 467
- setKeys() method 341
- setLocale() method 461
- setParamDataTypeJavaObj() method (deprecated) 305, 308, 492
- setParamDataTypeJDBC() method (deprecated) 305, 308, 492
- setParamIndex() method (deprecated) 305, 493
- setParamIOType() method (deprecated) 305, 493
- setParamName() method (deprecated) 305, 494
- setParamValue() method (deprecated) 305, 494
- setParticipantDefinition() method 468
- setRelationshipDefinition() method 468
- Setting
  - business object attribute 339, 343
  - business object attribute default value 341
  - business object contents 340
  - business object key attribute value 341
  - business object value in an array 357
  - business object verb 342
    - verb of child business object attribute 343
- setVerb() method 196, 281, 342, 346
- setVerbWithCreate() method 343
- setWithCreate() method 343
- Simple identity relationship 225, 226, 244, 263
  - child-level 273
  - defining 245, 246, 263, 273
  - defining Cross-Reference transformation 263
  - defining transformation rules 273
  - example of 225
  - main map 273
  - maintainChildVerb() 273, 290
  - maintainSimpleIdentityRelationship() 263, 448
  - parent map 273
  - participant type for 263
  - submap 274
- Single-map map table 61
- size() method 350, 358
- Source business object 3, 5, 185
  - adding to map 31, 34, 35
  - business object window 35
  - displaying 11, 19, 26, 60
  - testing 88
  - variable for 169
- Split transformation 18, 23, 37, 41, 52, 56, 84, 104
- Splitting strings
  - creating the parsed string 427
  - getting element at specified position 428

- Splitting strings (*continued*)
  - getting first element from string 428
  - getting last element from string 430
  - getting next element from string 430
  - getting number of elements in string 429
  - getting previous element from string 431
  - processing the parsed tokens into an object 430
  - resetting current position number 432
- SQL query 10, 203, 219, 299, 309
  - checking for more rows 206, 371, 385
  - executing 204, 300, 367, 368, 370, 383
  - prepared 209, 367
  - retrieving next row 206, 373
  - static 205, 368
- Standard toolbar (Activity Editor) 110
  - Copy 110
  - Cut 110
  - Delete 110
  - Help 110
  - Paste 110
  - Print Activity 110
  - Save Activity 110
- Standard toolbar (Map Designer) 25
  - displaying 24, 26
  - Find 72
  - New Map 31
  - Open Map from File 57
  - Open Map from Project 56
  - Print 74
  - Save Map to File 51
  - Save Map to Project 49
- Standard toolbar (Relationship Designer) 241
  - displaying 240, 242
  - New Participant 243
  - New Relation 243
  - Save Relation 244
- start\_server.bat file 167, 169
- Static lookup 285
- Static Lookup function block, example of using 158
- Static Lookup relationship 157
- Static relationship 253, 254
- Status bar (Activity Editor) 111
- Stored procedure
  - executing 211, 303, 368, 369, 370, 384
  - for relationship instance 248, 251
  - query result 211, 372, 373
- Stored-procedure parameter 213
  - creating object for 213, 305, 377, 487
  - in/out parameter type 213, 305, 378, 490, 493
  - index position 305, 489, 493
  - Java Object type 305, 488, 492
  - JDBC data type 305, 489, 492
  - mappings from Java Object to JDBC 215, 307
  - mappings from JDBC to Java Object 308
  - name 305, 491, 494
  - value 213, 305, 379, 491, 494
- String class 507
  - as stored-procedure parameter type 215, 308, 377
  - checking for valid data 345
  - converting to 398
  - converting to Boolean 392, 395
  - converting to Double 393, 396
  - converting to Float 394, 396
  - converting to Integer 394, 397
  - determining data type 390
  - getting attribute value 333
  - obtaining maximum value 351, 352
- String class (*continued*)
  - obtaining minimum value 353, 354, 355
  - setting attribute to 339
  - valid conversions 392
- String function block 141
  - Append Text 141
  - example of using 146
  - If 141
  - Is Empty 141
  - Is NULL 141
  - Left Fill 141
  - Left String 142
  - Lower Case 142
  - Object To String 142
  - Repeat 142
  - Replace 142
  - Right Fill 142
  - Right String 142
  - Substring by position 142
  - Substring by value 142
  - Text Equal 142
  - Text Equal Ignore Case 143
  - Text Length 143
  - Trim Left 143
  - Trim Right 143
  - Trim Text 143
  - Upper Case 143, 146
- String transformations 182
  - checking for blank 185
  - checking for null 185
  - converting to uppercase text 183
  - manipulating strings 183
- STRING\_TYPE constant 390
- stringReplace() method 434
- Strings
  - padding with specified character 433
  - replacing one pattern with another 434
- StringTokenizer class 427
- strInitiator built-in variable 176
- Submap transformation 37
- Submaps 43, 46, 198, 203
  - accessing code for 104
  - attribute comment for 52
  - calling 45, 195, 198, 199, 200, 202, 425
  - child business objects 44, 45
  - compiling 45, 85, 195, 199, 202
  - conditions 46, 198
  - creating 45, 195, 199, 202
  - defined 43
  - execution context and 190
  - Expression Builder and 200
  - identity relationships and 274, 281
  - key mapping for 23
  - many-to-one 201
  - multiple-cardinality 194
  - naming conventions 45
  - transformation code for 18
  - uses for 43
  - validating 56, 84
- sum() method 358
- summary, CwBidiEngine method 361
- swap() method 358
- Switch to Project (Relationship Designer) 239
- Synonym creation
  - editing 71
  - for automatic mapping 71
- System Manager 11

System Manager (*continued*)  
  compiling a map 85  
  Component menu 85, 244, 506  
  connection pools 204  
  Map Properties window 59, 186, 189, 216, 506  
  opening map from project in 56  
  relationship categories 254  
  starting Map Designer from 15  
  starting Relationship Designer from 237  
SystemException exception 322

## T

Table tab (Map Designer) 17, 19  
  adding business object 34  
  attribute transformation table 17, 74  
  business object pane 19, 24, 26, 34, 75, 173  
  business object variables 169  
  calling a submap 45  
  custom transformation 48  
  default display 24  
  deleting a transformation 75  
  deleting business object 75  
  joining attributes 39  
  moving attribute 38  
  output window 17  
  setting attribute value 37  
  specifying execution order 78  
  splitting attribute 41  
  temporary business object 173  
Temporary variables 171  
Test run 87  
  breakpoints 90, 93  
  creating test data 88  
  initial 88  
  pausing 90, 93  
  preparing for 88  
  starting 94  
  subsequent 90  
  viewing results 93  
toBoolean() method 392  
toDouble() method 393  
toFloat() method 393  
toInteger() method 394  
Tools menu (Activity Editor) 109  
  Check for Unmatched Delimiters 109  
  Edit Code 109  
  Expression Builder 110  
  Translate 109  
Tools menu (Map Designer) 27  
  Automatic Mapping 27  
  Business Object Designer 27  
  Map Designer 27  
  Process Designer 27  
  Relationship Designer 27  
  Reverse Map 27  
Tools menu (Relationship Designer) 242  
  Business Object Designer 242  
  Map Designer 242  
  Relationship Manager 242  
toPrimitiveBoolean() method 395  
toPrimitiveDouble() method 395  
toPrimitiveFloat() method 396  
toPrimitiveInt() method 397  
toString() method 179, 344, 345, 359, 398, 422  
Trace level 319, 504, 505  
Trace message 324, 502, 504, 506  
Trace message (*continued*)  
  adding 504  
  assigning trace level to 504  
  generating 505  
  setting trace level for 506  
trace() method 324, 502, 504  
Tracing 504, 506  
  code example 505  
  generating message 505  
  level for 504  
Transactions  
  beginning 217, 303, 365, 381  
  committing 217, 219, 303, 309, 366, 382  
  defined 216, 303  
  determining if in progress 219, 372, 386  
  explicit 216  
  implicit 216  
  managing 208, 216  
  rolling back 217, 303, 374, 387  
  scope 217  
Transformation code  
  auto-update mode 48, 104  
  checking 83  
  deleting 74  
  finding text in 72  
  generating 43  
  handling exceptions in 186  
  importing packages into 166  
  location of 84  
  missing 51  
  programming considerations 295  
  unmatched delimiters 83  
  viewing 64  
Transformation step 5, 18, 30, 35, 74, 84  
Transformations 6, 18, 36, 49, 173, 185  
  addressing attributes 169  
  checking code 83  
  checking completeness of 51  
  content-based logic 174  
  Context menu 27  
  Cross-Reference 18, 37, 47  
  Custom 18, 37, 47  
  date formatting 179  
  defining for relationships 257, 277  
  destination attribute 18  
  execution order 17, 56, 77, 84  
  in map definition file 50  
  introduction 5  
  Join 18, 36, 39  
  map document for 60  
  Move 18, 36, 38  
  relationship attributes 48, 52, 257  
  Reverse map 68  
  selecting 26  
  Set Value 18, 36, 37  
  source attribute 18  
  Split 18, 37, 41  
  standard 18, 36, 104  
  string 182  
  Submap 18, 37  
  validating 56, 84  
  validating source data 187  
  variables 171  
truncate() method 435  
try statement 101, 197, 297  
Type attribute property 507

## U

- UNKNOWN\_TYPE constant 390
- Update My Children function block 283
- UPDATE statement 208, 302, 368, 369, 371
- Update verb
  - conditionally set 285
  - foreignKeyXref() and 292, 293
  - maintainChildVerb() and 288, 289
  - maintainCompositeRelationship() and 276
  - maintainSimpleIdentityRelationship() and 266, 268, 269, 272
- updateMyChildren() method 284, 450, 485
- updateParticipant() method 483
- updateParticipantByInstance() method 484
- Upper Case function block, example of using 146
- UserMapMessages.txt message file 498, 499
- UserStoredProcedureParam class (deprecated) 11, 487, 494
  - constructor 487
  - getParamDataTypeJavaObj() 488
  - getParamDataTypeJDBC() 489
  - getParamIndex() 489
  - getParamIOType() 490
  - getParamName() 491
  - getParamValue() 491
  - method summary 487
  - setParamDataTypeJavaObj() 492
  - setParamDataTypeJDBC() 492
  - setParamIndex() 493
  - setParamIOType() 493
  - setParamName() 494
  - setParamValue() 494
- UserStoredProcedureParam() constructor (deprecated) 305, 487
- Utilities function block 143
  - Catch Error 143
  - Catch Error Type 143
  - Condition 143
  - Loop 143
  - Move Attribute in Child 144
  - Raise Error 144
  - Raise Error Type 144
- Utilities/Vector function block 144
  - Add Element 144
  - Get Element 144
  - Iterate Vector 144
  - New Vector 144
  - Size 144
  - To Array 144

## V

- validData() method 345
- Variable 169, 173
  - cwExecCtx 190, 425, 438, 439, 441, 442, 447, 449, 451
  - declaring 173
  - for business object 169
  - global 171, 173, 185
  - strInitiator 176
  - temporary 171
- Vector class 301
  - with executeStoredProcedure() 213, 368, 378
  - with nextRow() 206, 373
- Verb
  - defined 36
  - retrieving 336
  - setting 36, 53, 195, 196, 202, 285, 342, 444

Verb (*continued*)

- test run 88, 89
- verb-based logic 176
- Verb-based logic 176
- View menu (Activity Editor) 109
  - Content window 109
  - Design mode 109
  - GoTo 109
  - Library window 109
  - Preferences 109
  - Properties window 109
  - Quick view mode 109
  - Status Bar 109
  - Toolbars 109
  - Zoom In 109
  - Zoom Out 109
  - Zoom To 109
- View menu (Map Designer) 24, 26
  - Business Object Pane 19, 24, 26
  - Clear Output 17, 24, 26, 86
  - Diagram 20, 24, 26, 37
  - Output Window 17, 24, 26
  - Preferences 21, 27
  - Server Pane 19, 24, 26
  - Status Bar 17, 24, 27
  - Toolbars 24, 26
- View menu (Relationship Designer) 240, 242
  - Collapse Tree 242
  - Expand Tree 242
  - Participant Types 242, 243
  - Status Bar 240
  - Toolbar 240

## W

- Warning message 320, 503
- Web services
  - example of using in a map 160
  - exporting into Activity Editor 159
  - using in a map 159
  - using in Activity Editor 159

## Z

- Zero-length string 336







Printed in USA