SAS® 9.1.3 Java Metadata Interface: User's Guide

9.1.3

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2004. *SAS® 9.1.3 Java Metadata Interface: User's Interface* . Cary, NC: SAS Institute Inc.

SAS® 9.1.3 Java Metadata Interface: User's Interface

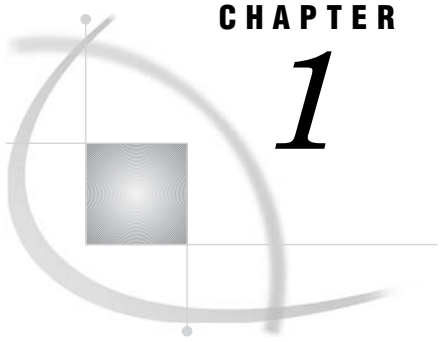
ISBN-13: 978-1–59047-523-2
ISBN-10: 1-59047-523-2

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, July 2004
2nd printing, November 2006

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

Contents



CHAPTER

1

Understanding the SAS Java Metadata Interface

About This Guide

This guide provides usage information about the SAS 9.1.3 Java Metadata Interface. Reference information about the interface is provided as class documentation that is shipped with the product. You can view a web-enabled version of the reference documentation at support.sas.com/rnd/gendoc/bi/api/.

Interface Overview

The SAS Java Metadata Interface provides a Java object interface to the SAS Metadata Server. The interface provides a way to access SAS metadata repositories through the use of client Java objects that represent server metadata. The interface consists of classes for

- connecting to the metadata server
- instantiating an object factory that creates Java objects to represent the SAS Metadata Model
- creating, reading, and writing Java metadata object instances on the client and propagating additions and changes to the SAS Metadata Server.

There are two implementations of the SAS Java Metadata Interface:

- a static version for visual applications that use only one Java Virtual Machine. Example applications include SAS Data Integration Studio and SAS Management Console plugins.
- a remote version for applications that have multiple tiers and use more than one Java Virtual Machine. Example applications include: SAS Report Studio, SAS Information Map Studio, and any application that uses SAS Foundation Services implementations.

The SAS Java Metadata Interface includes the following Java packages:

com.sas.metadata
: provides the static Java object interface to the SAS Metadata Server.

com.sas.metadata.impl
provides the implementation of the static interface to the SAS Metadata Server.

com.sas.metadata.remote
provides the remote Java object interface to the SAS Metadata Server.

com.sas.metadata.remote.impl
provides the implementation of the remote interface to the SAS Metadata Server.

The com.sas.metadata.remote packages are typically used in conjunction with com.sas.services.information package included with SAS Foundation Services software. The com.sas.services.information package provides a generic interface for interacting with heterogeneous data repositories, including SAS Metadata Repositories, Lightweight Directory Access Protocol (LDAP) repositories, and WebDAV repositories, from client applications. Using Information Service methods, a client can submit a single query that searches all available repository sources and returns the results in a "smart object" that provides a uniform interface to common data elements. The com.sas.services.information package is described in the SAS Foundation Services class documentation. SAS Foundation Services is a component of SAS Integration Technologies. Both the software and the class documentation are available from the SAS Installation Kit CD-ROM software media.

Software Installation and JRE Requirements

SAS Java Metadata Interface software is supported in UNIX and Windows host environments. You can install the software and class documentation from the SAS Software Installation Kit CD-ROM software media that is shipped with SAS 9.1.

The current release of the Java client software requires Java 2 Standard Edition Version 1.4 (JDK 1.4). The Java Runtime Environment can be obtained from the Third Party Software Components CD included in your SAS Software Installation Kit.

The javadoc can be viewed with Microsoft Internet Explorer Web Browser or Netscape Web Browser versions that support frames.

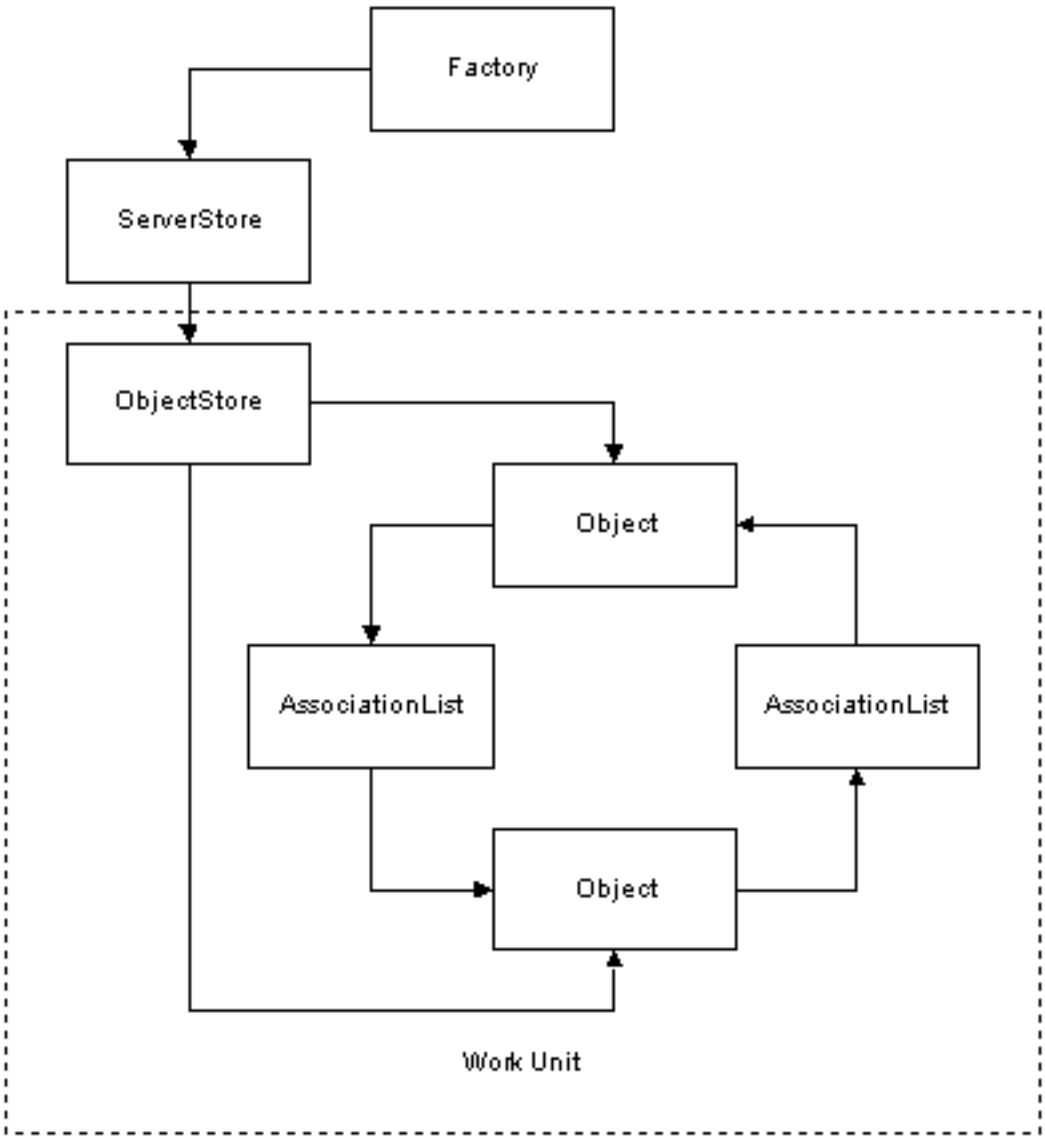
Understanding the Interface

The SAS Java Metadata Interface consists of

- an *object factory* for creating and controlling the lifecycle of objects in the client
- a *server store* for managing objects created by the object factory
- *object stores* that serve as work-unit containers for updating object instances and for grouping object instances that need to be persisted to the server as a unit
- Java objects for managing a metadata object's properties.

The object factory and server store represent the SAS Metadata Model in the client and provide an environment for managing Java objects that represent SAS metadata object instances.

The object store serves as a container or work unit for Java objects that users create to add metadata objects or to modify existing metadata objects on the metadata server. The following figure illustrates the relationship between the objects in an object store.



A SAS Open Metadata Interface metadata object is defined by two types of properties:

- a set of attributes that describe the characteristics of the object instance, including its name, description, the date it was created, and any unique characteristics
- association names that describe its relationships with other metadata objects.

Using the SAS Java Metadata Interface, you get and set a metadata object's attributes by creating a Java object representing its native type. A *native type* refers to one of the metadata types defined in the SAS namespace of the SAS Metadata Model. Information about associations is managed separately from information about attributes. You get and set associations by creating AssociationList objects. An AssociationList object stores information about the metadata objects related to a metadata object via a specific association name. To determine the associations defined for a specific metadata type, consult the "Alphabetical Listing of SAS Namespace Metadata Types" in the *SAS Open Metadata Interface: Reference*. Also see "Understanding Associations" in "Overview of the SAS Metadata Model and Model Documentation" in the reference. The alphabetical listing is available only in online versions of the reference. Look for it in *SAS Help and Documentation* or *SAS OnlineDoc*.

In the previous figure, the squares named 'Object' represent native objects and the squares named 'AssociationList' describe the relationships (associations) between the native objects. Every relationship in the SAS Metadata Model is a two-way association. That is, there are two sides to each relationship and each side has a name. For example, if the native objects in the illustration represented a PhysicalTable and a Column, the PhysicalTable object would have a Columns association to the Column object and the Column object would have a Table association to the PhysicalTable object.

See "Method Classes Summary" on page 17 for an overview of the method classes used to create the factory, stores, and other objects.

See "Overview" on page 5 for instructions about how to write a SAS Java Metadata Interface client that reads and writes metadata.

For documentation describing the classes and methods, see the SAS Java Metadata Interface at support.sas.com/rnd/gendoc/bi/api/.

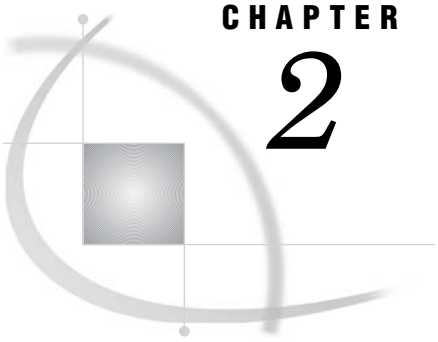
Important Terms

A *metadata type* is a template that models the metadata for a particular kind of object.

A *metadata object* is an instance of a metadata type, such as the metadata describing a particular table or column.

A *namespace* is a group of related metadata types and their properties. Names are used to partition metadata into different contexts.

In the current SAS release, the SAS Java Metadata Interface provides interfaces for metadata types defined in the SAS namespace of the SAS Open Metadata Interface. The SAS namespace contains metadata types describing the most commonly used application elements. These metadata types are described in "Alphabetical Listing of SAS Namespace Metadata Types."



CHAPTER

2

Using the SAS Java Metadata Interface

Overview

SAS Java Metadata Interface software is provided to make it as simple as possible to use the functionality of the SAS Metadata Server in a Java program. Using the SAS Java Metadata Interface, you can write Java client programs that make use of SAS Open Metadata Architecture XML metadata objects as if they were Java objects. There is no need to learn SAS Open Metadata Interface method calls or XML, although users must be familiar with the metadata types in the SAS Metadata Model and the flags and options supported by SAS Open Metadata Interface IOMI Class methods. For information about the metadata types defined in the SAS Metadata Model, see "Alphabetical Listing of SAS Namespace Metadata Types" in the *SAS Open Metadata Interface: Reference*. For information about flags and options, see the "Methods for Reading and Writing Metadata (IOMI Class)" section of the reference. The alphabetical listing is available only in online versions of the reference. Look for it in *SAS Help and Documentation* or *SAS OnlineDoc*.

The interface adheres to Java distributed programming standards such as CORBA and JDBC, so that when you write a Java client program that uses the SAS Metadata Server—whether that program is an applet, a stand-alone application, a servlet, or an enterprise JavaBean—you can focus your attention on exploiting the features of the SAS Metadata Server rather than figuring out how to communicate with it.

The SAS Java Metadata Interface includes all the tools that you need to work with the SAS Metadata Server from a Java client. Previous knowledge of distributed programming standards is not required, nor are you required to license any third-party software.

Creating a SAS Java Metadata Interface Client

This section introduces the steps necessary to construct and execute a SAS Java Metadata Interface client that reads and writes metadata.

The first step in developing and running a client program is to make sure that you have access to a properly configured SAS Metadata Server. You have a properly configured SAS Metadata Server if you performed a standard SAS installation, used the SAS Configuration wizard to configure your SAS Business Intelligence environment, and followed the instructions in the *SAS Intelligence Platform: System Administration Guide* to perform optional customizations.

After the metadata server has been configured, you can begin developing a SAS Java Metadata Interface client that uses it. All SAS Java Metadata Interface clients access a SAS Metadata Server using the following steps:

1. Instantiate an object factory that defines Java objects representing the SAS Metadata Model.
2. Invoke event handling and messaging mechanisms.
3. Connect to the metadata server referencing the appropriate SAS Open Metadata Interface method class.
4. Create Java metadata object instances representing SAS Open Metadata Interface metadata objects and get and set attributes and associations as needed.
5. Issue the UpdateMetadataAll method to persist the changes to the metadata server.

To get started, you can put together a simple client application by composing the examples given for each step. Then you can continue to read the additional documentation in this user's guide and learn about Java client programming for the SAS Metadata Server in greater detail.

Instantiating an Object Factory and Connecting to the Server

This topic contains an example of the SAS Java Metadata Interface calls necessary to instantiate an object factory and to connect to the SAS Metadata Server.

You create an object factory by instantiating the *MdObjectFactory* class. This class contains all of the methods necessary to create Java metadata objects and to invoke Java event handling and messaging mechanisms.

You create a connection to the server using a method from the *MetadataWorkspace* class. The MetadataWorkspace class provides task-oriented methods for connecting to the metadata server. For example,

- if the purpose of the SAS Java Metadata Interface client is to read and write metadata, you use the makeOMRConnection method. This method implements the SAS Open Metadata Interface IOMI class.
- if the purpose of the client is to manage the metadata server, you use the makeIServerConnection method. This method implements the SAS Open Metadata Interface IServer class.
- if the purpose of the client is to make an authorization request of the authorization facility, which is part of the SAS Metadata Server, then you use the makeISecurityConnection method. This method implements the SAS Open Metadata Interface ISecurity class.

Note: In the current release, the SAS Java Metadata Interface provides an interface to IOMI methods for SAS namespace metadata types. That is, Java methods are provided for reading, writing, and updating metadata objects representing application elements. To read or write metadata types in the REPOS namespace (repository objects) or to issue methods from the SAS Open Metadata Interface ISecurity and IServer method classes, you can pass XML-formatted method calls via the Java implementation for the DoRequest method. There are no Java methods for these

classes. The Java interface for the DoRequest method is in the SAS Java Metadata Interface *MetadataUtil* class. △

Example of Connecting to the Metadata Server

An object factory is instantiated one time only for a client. Depending on the task that you want to perform, you might need to disconnect and reconnect to the metadata server using a different connection method. The following example creates a single connection using the makeOMRConnection method.

Here is the connection code:

```
/**
 * This statement instantiates the object factory.
 */
private MdObjectFactory mdFact = MdObjectFactory.getInstance();

/**

 * The following statements define variables for server connection properties,
 * instantiate the MetadataWorkspace class, issue a makeOMRConnection method,
 * and check exceptions if there is an error connecting.
 */
public boolean Example1()
{
   String serverName = "server_machine_name";
   String serverPort = "8561";
   String serverUser = "username";
   String serverPass = "password";
   MetadataWorkspace workspace = MetadataWorkspace.getWorkspace();

   try
   {
   // This statement passes server connection properties to the makeOMRConnection
   // method.
     workspace.makeOMRConnection(serverName, serverPort , serverUser, serverPass);

   // The following statements define error handling and error reporting messages:
   }catch (MdException e)
   {
      Throwable t = e.getCause();
      if(t != null)
      {
         String ErrorType = e.getSASMessageSeverity();
         String ErrorMsg = e.getSASMessage();
         if(ErrorType == null)
         {
          // If there is no SAS server message, write a Java/CORBA message.
         }else{
          // If there is a message from the server:
           System.out.println(ErrorType + ": " + ErrorMsg);
         }
         if(t instanceof org.omg.CORBA.COMM_FAILURE)
         {
           // If there is an invalid port number or host name:
```

```
                System.out.println(e.getLocalizedMessage());
            }else if(t instanceof org.omg.CORBA.NO_PERMISSION)
            {
              // If there is an invalid user ID or password:
                System.out.println(e.getLocalizedMessage());
            }
         }else{
            // If we cannot find a nested exception, get message and print.
            System.out.println(e.getLocalizedMessage());
         }
         // If there is an error, print the entire stack trace.
         e.printStackTrace();
         return false;
      }catch (RemoteException e)
      {
         // Unknown exception.
         e.printStackTrace();
         return false;
      }

      // Set the log and system output locations for the log and output streams.
      try
      {
         Java.io.FileOutputStream mylogFile = new Java.io.FileOutputStream("Testlog.log");
         Java.io.FileOutputStream myoutputFile =
            new Java.io.FileOutputStream("TestOutput.log");
         Util.setLogStream(mylogFile);
         Util.setOutputStream(myoutputFile);
      }catch (Java.io.FileNotFoundException e)
      {
         // If an error occurs during logging setup, then exit.
         e.printStackTrace();
         return false;
      }

      // If no errors occur, then a connection is made.
      return true;
   }
```

From this example, we have

- a factory in which to create Java objects
- log and output location definitions
- an available connection to the metadata server.

We can now get information about repositories defined on the metadata server and create metadata object instances.

Getting Information About Repositories

Before you can read or write metadata, you must identify the repositories registered on a given metadata server. You will need to be familiar with the repository identifiers in order to indicate which repository to access. You can list the repositories that defined on a metadata server by using the getRepositories method. The getRepositories method exists in the *MetadataUtil* class.

Here is sample code that issues a getRepositories request:

```
public List Example2()
{
   try{
    // Print a descriptive message about the request.
    System.out.println("The Repositories contained on this SAS Metadata Server are: ");
    // Get a list of repositories.
    List reposList = MetadataUtil.getRepositories();
    Iterator iter = reposList.iterator();
    while(iter.hasNext())
      {
         CMetadata repository = (CMetadata)iter.next();
         Util.printOutputln("Repository: "
                           + repository.getName()
                           + ", "
                           + repository.getFQID());
      }
      CMetadata mainRepos = (CMetadata)reposList.get(0);
      Util.printOutputln("\n");
      return reposList;
   }catch (MdException e)
   {
      e.printStackTrace();
   }
   return new Vector(1);
}
```

Here is an example of the output that might be returned by the request:

```
The Repositories contained on this metadata server are:
Repository: DW: Demo Warehouse, A0000001.A5K2EL3N
Repository: ENV: Demo Warehouse, A0000001.A50TC1Z2
Repository: ENV: Codegen Test Env, A0000001.A5V79M59
Repository: DW: Codegen, A0000001.A5SHTOLR
```

The two-part number in each line is the repository identifier. The first part (A0000001) is the repository manager identifier and is the same for all repositories. The second part is the unique repository ID. This is the identifier that we will use to read and write metadata.

The CMetadata interface in this example is the base interface used to describe all metadata objects. These method parameters take in a CMetadata object to allow for the methods to be used with any SAS Open Metadata Interface metadata object.

Creating Objects

You can create objects by using the methods in the *MdObjectFactory* class. You must create a Java object instance for every new metadata object and for every existing metadata object that you want to update or delete in a SAS Metadata Repository. You must also create an object store in which to hold the objects. The object store maintains a list of the objects that need to be persisted to the metadata server at one time.

The following code creates a new PhysicalTable metadata object instance, a new Column metadata object instance, and a new TextStore metadata object instance; it then creates associations between these object instances. After the metadata objects are created, they are persisted to repository A0000001.A5K2EL3N (Demo Warehouse).

Notes:

- You can identify the repository in which to persist an object by specifying its repository identifier in the createComplexMetadataObject method. Or, you can use methods from the CMetadata interface. The CMetadata interface enables you to determine the identifier of a target repository and reference it in the createComplexMetadataObject method as a variable.
- Because these are new metadata objects, they will be assigned metadata object identifiers when they are persisted to the SAS Metadata Server. A request that creates Java objects to represent existing metadata objects would need to determine their metadata object instance identifiers prior to persisting the updates. For more information, see "Getting and Updating Existing Objects" on page 11.

```
/**
 * This is a good example of how a wizard-style
 * user interface would utilize the MdObjectFactory classes.
 *
 *
 * @param Repository CMetadata Object with id of form:  A0000001.A5K2EL3N
 */
public void Example3(CMetadata Repository)
{
   if(Repository != null)
   {
     try
      {
       // We have a Repository object.
       // We use the reposFQID method to get its fully qualified ID.
       String reposFQID = Repository.getFQID();

       // The reposFQID method returns the 17-character repository identifier.
       // We need the unique, 8-character repository ID to create an object.
       // We use the substring method to get the unique portion of the identifier.
        String ReposID =
        reposFQID.substring(reposFQID.indexOf('.') + 1, reposFQID.length());

       // Now we create an object store to hold all of our objects.
       // This will be used to maintain a list of objects to persist to the server.
        MdObjectStore myStore = MdObjectFactory.createObjectStore();

      // We create a PhysicalTable object named "TableTest".
      PhysicalTable myTable =(PhysicalTable)MdObjectFactory.createComplexMetadataObject
                                  (myStore,
                                   null,
                                   "TableTest",
                                   MdObjectFactory.PHYSICALTABLE,
                                   ReposID);

      // We create a Column object named "ColumnTest".
      Column myColumn = (Column)MdObjectFactory.createComplexMetadataObject
                                   (myStore,
                                    null,
                                    "ColumnTest",
                                    MdObjectFactory.COLUMN,
                                    ReposID);
```

```
        // We set attributes of the column.
        myColumn.setColumnName("MyTestColumnName");
        myColumn.setSASColumnName("MyTestSASColumnName");
        myColumn.setDesc("This is a description of a column");

      // We use the get"AssociationName"() method to associate the column with the
      // table. This method creates an AssociationList object for the table object.
      // We could have specified get"AssociationName"(false)" here, but this method
      // does not go to the server for temporary objects. If the object already
      // existed, specifying the "false" flag will tell it not to go to the server
      // to get the list of columns. The Add(MetadataObject) method adds myColumn
      // to the AssociationList.

         myTable.getColumns().add(myColumn);

      // We create a note for the column named "NoteTest".
      TextStore myNote = (TextStore)MdObjectFactory.createComplexMetadataObject
                                   (myStore,
                                    null,
                                    "NoteTest",
                                    MdObjectFactory.TEXTSTORE,
                                    ReposID);

      // We use the set"AttributeName" method to set stored text with the note.

        myNote.setStoredText("I have some information about the column");

      // We associate the note with the column.
        myColumn.getNotes().add(myNote);

      // We issue an update request on an object in the object store.
      // When one object is updated, all objects in the store's change list
      // get persisted to the server.

         myTable.updateMetadataAll();

      // Now we need to clean up the objects, if we are no longer using them.
         myStore.dispose();
      }catch (MdException e)
      {
         e.printStackTrace();
      }
   }
}
```

For more information about object stores and AssociationList objects, see "Interface Overview" on page 1.

Getting and Updating Existing Objects

In order to update an existing metadata object, you must know its metadata object instance identifier. The SAS Java Metadata Interface provides several ways for getting information about existing objects. This section provides an example of one way you

might read information about the metadata objects created in "Creating Objects" on page 9. The example uses the getMetadataObjectsSubset method from the *MetadataUtil* class.

The getMetadataObjectsSubset method gets a list of metadata objects in the repository of a requested type and enables you to filter the request with SAS Open Metadata Interface templates and XMLSELECT statements. In the example that follows, flags and templates are used to retrieve all PhysicalTable objects named "TableTest," their associated Column and Note metadata objects, and specific attributes of all of the objects.

Note: The <TEMPLATES> and <XMLSELECT> elements submit input XML strings to the metadata server as a string literal (a quoted string). To ensure that the string is parsed correctly, you must escape any additional double quotation marks specified in the input string, such as those used to denote XML attribute values, to indicate that they should be treated as characters. In this example, additional quotation marks are escaped by using a backslash (\) character. For example, ""is specified as \"\". △

The objects are returned in an object store and are editable.

```
/**
 * This method reads the newly created objects back from the server.
 * @param repository1 identifies the repository from which to read our objects.
 */
public void Example4(CMetadata repository1)
{
  if(repository1 != null)
  {
    try
    {
     // First we create an MdObjectStore as a container for all the objects
     // we will create/read/persist to the server as one collection.
     MdObjectStore myStore = MdObjectFactory.createObjectStore();

     //  The following statements define GetMetadataObjectsSubset options strings.
     //  These XML strings are used in conjunction with SAS Open Metadata Interface
     //  flags. The <XMLSELECT> element specifies filter criteria. The <Templates>
     //  element specifies the metadata properties to be returned for each object.
     //  Note the \ (backslashes) used to escape the quotation marks.
     String sOptions = "<XMLSELECT Search=\"@NAME='TableTest'\"/>"+
                       "<TEMPLATES><PhysicalTable Id=\"\" Name=\"\" Desc=\"\">"+
                       "<Columns/></PhysicalTable>"+
                       "<Column Id=\"\" Name=\"\" Desc=\"\"><Notes/></Column>" +
                       "<TextStore Id=\"\" Name=\"\" Desc=\"\" StoredText=\"\"/>"+
                       "</TEMPLATES>";
     //  The following statements go to the server with a fully qualified repository
     //  ID, specify the type of object we are searching for (MdObjectFactory.
     //  PHYSICALTABLE), and invoke the OMI_XMLSELECT, OMI_TEMPLATE, and
     //  OMI_GET_METADATA flags. OMI_GET_METADATA tells the server to get all of
     //  the attributes specified in the template for each object that is returned.
     List PhysicalTableList = (MetadataUtil.getMetadataObjectsSubset(myStore,
                                                    repository1.getFQID(),
                                                    MdObjectFactory.PHYSICALTABLE,
                                                    MetadataUtil.OMI_XMLSELECT |
                                                    MetadataUtil.OMI_TEMPLATE |
                                                    MetadataUtil.OMI_GET_METADATA,
                                                    sOptions ));
```

```
   Iterator iter5 = PhysicalTableList.iterator();
   while(iter5.hasNext())
   {
      PhysicalTable ptable = (PhysicalTable)iter5.next();

      Column columnTest = null;
      TextStore noteTest = null;

      // We get the list of columns for this table.
      AssociationList columns = ptable.getColumns();

      // Then get individual columns.
      for(int i=0; i < columns.size(); i++)
      {
         columnTest = (Column)columns.get(i);
         if(columnTest != null)
         {
            // We now have a column, and request to get its notes.
            AssociationList columnNotes = columnTest.getNotes();
            for(int j=0; j < columnNotes.size(); j++)
            {
               if(columnNotes.size() > 0)
               {
                  noteTest = (TextStore)columnNotes.get(0);
                  if(noteTest != null)
                  {
                  // We now have a valid note, and request to print its attributes.
                      System.out.println("TextStore Object: " +
                                         noteTest.getName() +
                                         ", " +
                                         noteTest.getFQID() +
                                         ", " +
                                        noteTest.getStoredText() );
                  }
               }
            }
         }
      }// end for columns

      // We now have a table, a column, and a note that we can update.
      // These statements modify the descriptions of the three objects.

      ptable.setDesc("The description of the table");
      columnTest.setDesc("The description of the column");
      noteTest.setDesc("The description of the note");

      //This statement persists the objects to the server.
      ptable.updateMetadataAll();
   }

    // We have completed our updates so we will dispose of the objects.
   myStore.dispose();

}catch (MdException e)
```

```
      {
         e.printStackTrace();

      }

   }
}
```

Here is the output of the code:

```
TextStore Object: NoteTest, A5K2EL3N.AN0000I3, I have some information
about the column
```

The output prints the modified note's metadata type (TextStore), its object name (NoteTest), its object instance identifier (A5K2EL3N.AN0000I3), and the content of its StoredText attribute ('I have some information about the column').

The XMLSELECT option is described in detail in the *SAS Open Metadata Interface: User's Guide*. See "Filtering a GetMetadataObjects Request" in "Querying All Objects of a Specified Metadata Type." For more information about templates, see "Using Templates" in the guide.

Deleting Objects

This is an example of how to delete metadata objects. As when updating objects, you must populate Java objects representing the server objects on the client before you can delete them. When you delete an object, all of its dependent objects will automatically be deleted as well. A dependent object is an object that has a 1:1 cardinality with the specified object and cannot exist independently of the object. An example of a dependent object is a Column. A Column object cannot exist in the SAS Metadata Model independently of some type of table object.

In this example, we use the getMetadataObjectsSubset method to get the objects that we created and updated in "Creating Objects" on page 9 and in "Getting and Updating Existing Objects" on page 11 and we use the deleteMetadataObjects method to delete them. The getMetadataObjectsSubset method is from the *MetadataUtil* class. The deleteMetadataObjects method is from the *MdObjectFactory* class.

```
/**
   * Delete the objects we just created in repository1.
   * @param repository1
   */
  public void Example6(CMetadata repository1)
  {
    if(repository1 != null)
    {

     try
     {

        MdObjectStore myStore = MdObjectFactory.createObjectStore();
        //  The following statements define GetMetadataObjectsSubset options
        //  strings. These XML strings are used in conjunction with SAS Open
        //  Metadata Interface flags. The <XMLSELECT> element specifies filter
        //  criteria. The <Templates> element specifies the metadata properties
        //  to be returned for each object from the server.
        String sOptions = "<XMLSELECT Search=\"@NAME='TableTest'\"/>"+
```

```
                        "<TEMPLATES><PhysicalTable Id=\"\" Name=\"\" Desc=\"\"/>"+
                        "</TEMPLATES>";
          // This statement creates a deleteTemplate object.
          String deleteTemplate = "<TEMPLATES><PhysicalTable Id=\"\" Name=\"\">"+
             "<Columns/><Notes/></PhysicalTable>"+
             "<Column><Notes/></Column></TEMPLATES>";

          //  The following statements go to the server with a fully qualified
          //  repository ID, specify the type of object we are searching for
          //  (MdObjectFactory.PHYSICALTABLE), and invoke the OMI_XMLSELECT,
          //  OMI_TEMPLATE, and OMI_GET_METADATA flags. OMI_GET_METADATA tells the
          //  server to get all of the attributes specified in the template for
          //  each object returned.

          List PhysicalTableList = (MetadataUtil.getMetadataObjectsSubset(myStore,
                                              repository1.getFQID(),
                                              MdObjectFactory.PHYSICALTABLE,
                                              MetadataUtil.OMI_XMLSELECT |
                                              MetadataUtil.OMI_TEMPLATE |
                                              MetadataUtil.OMI_GET_METADATA,
                                              sOptions ));
         // The following statements remove the objects returned by the preceding
         // query from the client and from the server. The code loops through the
         // list of objects and prints the name of each object before deleting it.
         // An event is sent to all object stores to tell them to delete the objects,
         // and to notify their users of a change in the store.

         Iterator iter5 = PhysicalTableList.iterator();
          while(iter5.hasNext())
          {
             PhysicalTable ptable = (PhysicalTable)iter5.next();
             Set assocNames = ptable.getAssocs().keySet();
             Iterator iter9 = assocNames.iterator();
             System.out.println("PhysicalTable: Associations");
             while(iter9.hasNext())
             {
                System.out.println((String)iter9.next());
             }
             MdObjectFactory.deleteMetadataObjects(ptable,deleteTemplate);
          }
           myStore.dispose();

       }catch (MdException e)
    }
  }
```

See SAS Java Metadata "Interface Overview" on page 1 to learn more about the object factory and methods for reading and writing metadata. For an executable version of this example, and the examples in Creating Objects, Getting and Updating Existing Objects, and a few additional examples, see "Sample Program" on page 28.

CHAPTER

3 Understanding the Method Classes

Method Classes Summary

A SAS Java Metadata Interface client that reads and writes metadata objects references the following com.sas.metadata method classes.

Table 3.1 com.sas.metadata Method Classes

Class Name	Description
MdObjectFactory	Instantiates the Java object factory and contains methods for creating and deleting Java objects representing server metadata. In the remote version, this class is called MdFactoryImpl.
MetadataWorkspace	Contains methods for connecting to the server. In the remote version, this class is called MdOMRConnection.
CMetadata	Specifies the base interface used to describe SAS Open Metadata Interface objects.
MdEvent	Contains event handling methods.

Class Name	Description
MetadataUtil	Contains methods for getting and setting server metadata objects and their attributes. In the remote version, this class is called MdOMIUtil.
AssociationList	Contains methods for defining and maintaining associations.
MdObjectStore	Specifies the container for created/modified metadata objects.
MdServerStore	Specifies the main object cache for interactions with the metadata server.
Util	Contains generic methods, such as debugging methods. In the remote version, this class is called MdUtil.

Working with the MdObjectFactory Class

The MdObjectFactory class instantiates the Java object factory and provides methods for creating and deleting Java objects, for invoking the SAS Java Metadata Interface event handling interface, and for deleting the object factory.

Instantiating the Object Factory

The object factory is instantiated one time only for a SAS Java Metadata Interface client before any other tasks are performed by using the getInstance method:

```
MdObjectFactory objf = MdObjectFactory.getInstance();
```

Creating Java Objects

After you have instantiated the object factory and connected to the server (using the makeOMRConnection method of the MetadataWorkspace class), you can then use the methods in the MdObjectFactory class to create objects on the client. MdObjectFactory provides two basic methods for creating objects:

createComplexMetadataObject
: creates a complex object that stores information about both a metadata object's attributes as well as its potential associations. You can use this method to create an object representing a new or existing object that will be persisted to the server.

createSimpleMetadataObject
: creates a simple object that consists only of a metadata object's attributes. Simple objects are used primarily to store information that is read to the client from the server and that cannot be persisted to the server. In order to edit information returned from the server, you must create a complex version of the object.

The following are examples of the createComplexMetadataObject method. A method call that creates a complex object describing a new metadata object has the form:

```
MdObjectFactory.createComplexMetadataObject(myNewObjectName,
                                            metadata_type,
                                            8char_target_repository_identifier)
```

A method that creates a complex object representing an object that already exists on the metadata server has the form:

```
MdObjectFactory.createComplexMetadataObject(myNewObjectName,
                                            metadata_type,
                                            identifier_of_existing_metadata_object)
```

You can obtain the identifiers of all repositories registered on a metadata server by using the getRepositories method of the MetadataUtil class. You can obtain the identifier of an existing metadata object instance by using one of the getMetadataObject methods of the MetadataUtil class. For more information about repository and metadata object instance identifiers, see "Identifying Metadata" in the "Methods for Reading and Writing Metadata" section of the *SAS Open Metadata Interface: Reference*.

Invoking the Event Handling Interface

The event handling interface is implemented by referencing the addMdObjectFactoryListener method. The addMdObjectFactoryListener method is instantiated one time only, either directly before or after the server connection is made. The event handling interface provides action events for all read, write, and delete calls in the SAS Java Metadata Interface client. It also implements internal messaging between objects in an object store and between object stores in the server store to ensure consistency between the objects that are persisted to the server.

The event handling interface operates as follows: the object store that performs a write action to the server sends action events to all other object stores, indicating which objects have changed. The other object stores can either veto the action or automatically update their objects to match the object store that was written to the server.

If a listener is added, it must be removed at the end of use.

Deleting Objects

To delete a metadata object from the metadata server, you must create an object representing it in the SAS Java Metadata Interface client and then delete both the server and client objects by calling the deleteMetadataObject method of the MdObjectFactory class. Calling this method removes the object from the server and clears its object store locally.

A new object that was created on the client and persisted to the server can be deleted from its object store by calling the CMetadata delete method. The CMetadata delete method marks the client object as deleted, such that it will be removed the next time the UpdateMetadataAll method is called on the object store.

The object store created to hold the client object will remain available to the client until it is disposed of using the MdObjectStore.dispose method. After the dispose method is called, all child stores and objects contained within this object store will be removed from memory.

Deleting the Object Factory

Use the MdObjectFactory dispose method to remove the object factory from memory before closing the client application. The method will also remove any object stores that were not removed by another means.

Working with the MetadataWorkspace Class

The MetadataWorkspace class contains methods for connecting to and disconnecting from the SAS Metadata Server. Before you can issue a MetadataWorkspace method, you must instantiate a workspace as follows:

```
MetadataWorkspace workspace = MetadataWorkspace.getWorkspace();
workspace.makeOMRConnection(serverName, serverPort, serverUser,
  serverPassword);
```

A SAS Java Metadata Interface client that reads and writes metadata uses the makeOMRConnection method to connect to the metadata server. It disconnects from the server using the closeOMRConnection method.

The MetadataWorkspace class also provides methods for connecting to the server using the SAS Open Metadata Interface IServer and ISecurity method classes. The IServer method class provides methods for administering SAS metadata repositories and the metadata server. The ISecurity class contains methods for defining authorization requests for the SAS Open Metadata Architecture Authorization Facility. The SAS Java Metadata Interface does not provide wrapper methods for IServer and ISecurity methods in the current release; however, XML-formatted IServer and ISecurity method calls can be passed to the metadata server through the DoRequest method of the MetadataUtilities class.

Working with the CMetadata Class

The CMetadata class is the intermediate interface used to describe all metadata objects, such as a PhysicalTable, Column, Person, or a LogicalServer. The CMetadata interface also contains the basic attributes for all metadata objects, such as Name, Description, FQID, MetadataCreated time, and MetadataUpdated time. All metadata objects inherit these attributes. They also inherit the routines used to get and set these attributes. For example, routines such as getName and setName or getDesc and setDesc are all inherited from CMetadata.

Other frequently used CMetadata methods are summarized in the following table.

Table 3.2 Frequently Used CMetadata Methods

Method Name	Description
delete	Marks an object as deleted in its parent object store and does not allow for retrieval of that object.
dispose	Removes all links to an object and clears it from memory.
getCMetadataType	Returns the metadata type of an object.
getMdObjectAssociation/setMdObjectAssociation	Get or set a specific association for an object on the client.
getObjectStore	Gets the object store associated with a named object.

Method Name	Description
getRepositoryID	Returns an object's repository identifier.
UpdateMetadataAll	Sends new and modified objects to the server.

Working with the MdEvent Class

An event is a notification of a change in a object store, object, or the state of an object. Events for a store notify the creation, deletion, and modification of objects as a collection. These are the results of a client-initiated server operation such as creation, modification, or deletion of objects on the server. Object events provide notification of modification of an object's attributes or associations. Usually these events are generated from the object being modified external to the object's store.

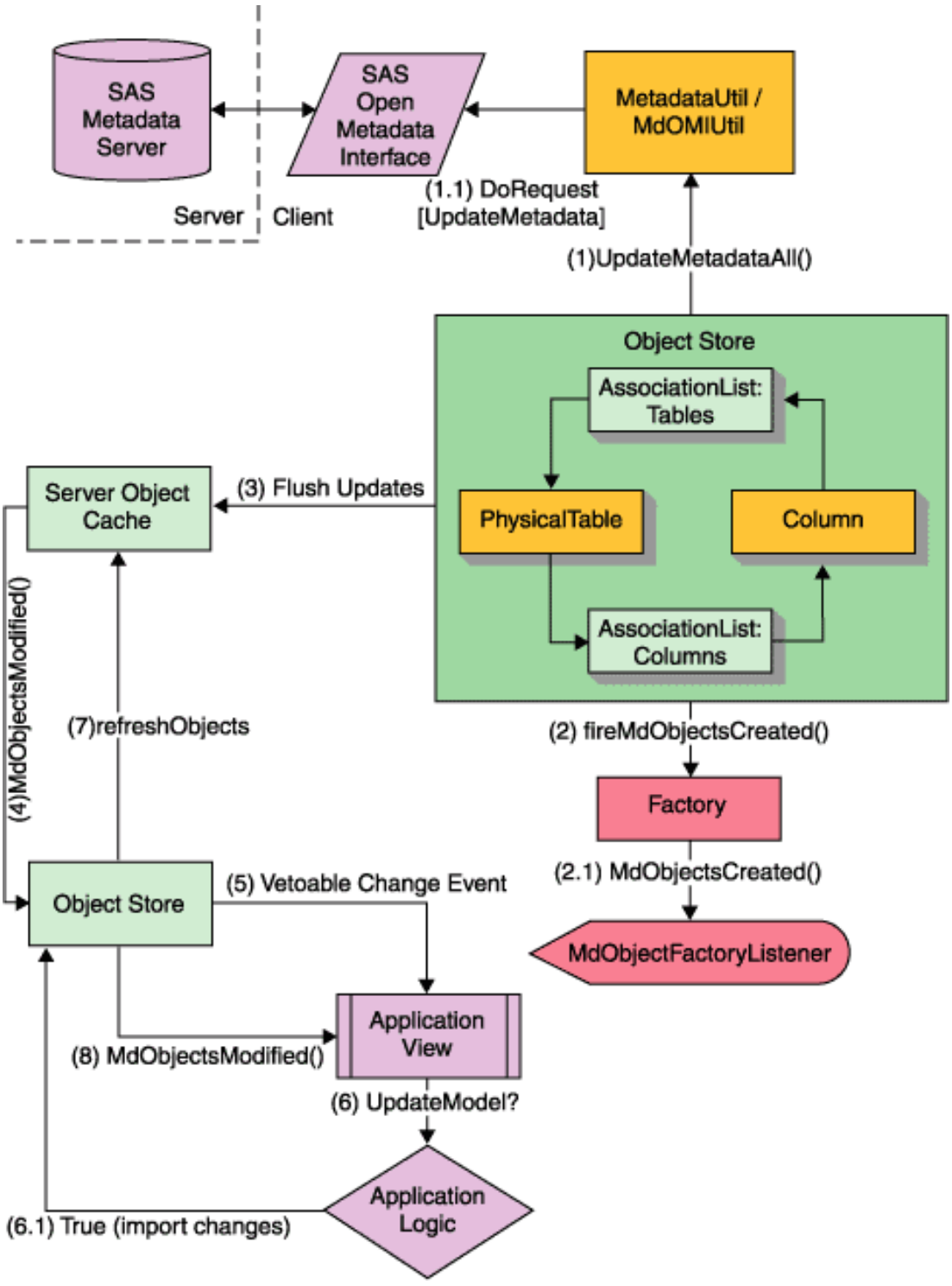
There are two types of events: primary events and secondary events. *Primary events* are generated when objects are added to the metadata server, when objects are modified on the client and persisted to the metadata server, and when objects are deleted on the metadata server. Primary events are sent to object stores from the server from the operation in MetadataUtil and the object store handles the events through its listeners.

Primary events are typically the result of an UpdateMetadataAll or a DeleteMetadataObject method call. For example, when you issue a MdObjectFactory.deleteMetadataObject(myMetadataObject) call, the client sends a DeleteMetadata method call to the metadata server and the metadata server returns a list of affected associated objects. This list of associated objects is processed in the factory and a Delete event is sent to all object stores to notify them of the repercussions of the deletion.

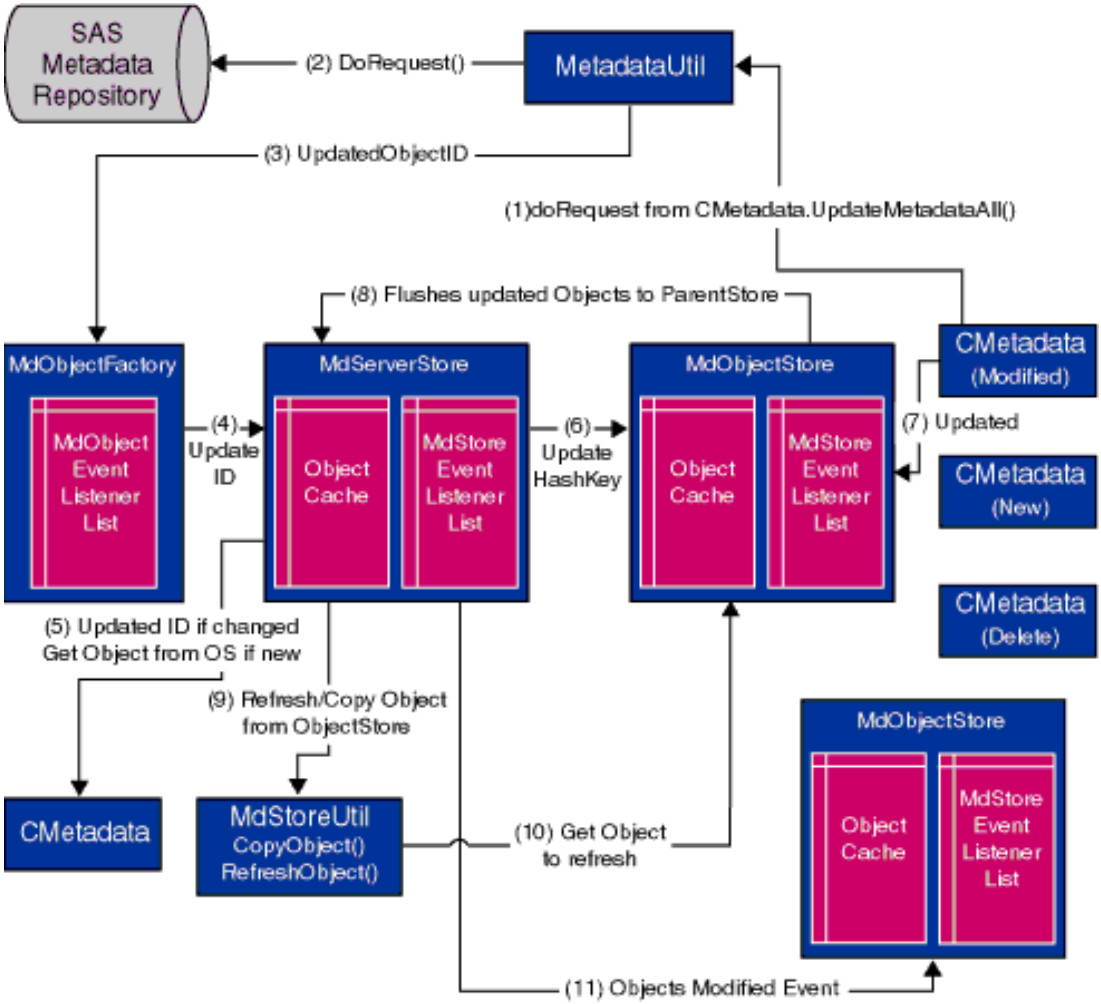
When objects are created on the metadata server, the events are generated in much the same way. The client sends input XML to the metadata server, and the server returns output XML. The client compares the identifiers in the client and server XML and changes the object identifiers on the client to match the newly created server objects. These objects are updated in all object stores on the client. Created events are generated in each store for the objects in that store that were persisted to the server and have new identifiers.

Secondary events are events that occur in the client. These are sometimes referred to as vetoable change events. A VetoableChangeEvent is generated by an object store that is to receive updated objects from its parent store. This event allows the application to "veto," or disallow, the update of the objects. This is useful when an application has more than one view of the metadata and you want to override local changes. It is also useful for determining the objects involved in a change.

The following figure illustrates the object store update/create event cycle.



The following figure illustrates the events scenario and write logic.



A description of each step in the second table follows:

1 An updateMetadataAll() method is called on an object, such as myTable.updateMetadataAll().

2 The SAS Java Metadata Interface calls the SAS Open Metadata Interface doRequest() method, which sends input XML to the server and returns output XML.

3 The SAS Java Metadata Interface retrieves the new object ID from the output XML.

4 Before objects are persisted to the server, they have temporary identifiers on the client. In this step, the SAS Java Metadata Interface updates the object's temporary identifier to match the identifier assigned by the server.

5 The SAS Java Metadata Interface updates the object's associations to reflect the new identifier.

6 The object's identifier is updated in any existing child stores.

7 The object store event listener fires an ObjectModified event to report the modifications to the server store.

8 The updated object is flushed to its parent store.

9 The updated object is then copied into the parent store, so that the child store and the parent store contain the same object.

10 The object is retrieved from its object store using the new object identifier.

11 The server store event listener fires an ObjectsModified event to notify other object stores of the updates.

Working with the MetadataUtil Class

The MetadataUtil class provides wrapper methods for methods in the SAS Open Metadata Interface IOMI class. MetadataUtil methods enable you to get the properties of existing metadata objects on the SAS Metadata Server. You can then create Java objects representing the SAS Open Metadata Interface metadata objects and get and set additional or modified properties.

The MetadataUtil class also provides AddMetadata, UpdateMetadata, and DoRequest methods. The AddMetadata and UpdateMetadata methods enable you to pass XML-formatted metadata property strings to update metadata objects on the server instead of creating Java objects. The DoRequest method enables you to pass XML-formatted IServer and ISecurity class methods from a SAS Java Metadata Interface client. Currently, the DoRequest method is the only Java interface for passing methods from these classes.

The following table summarizes the basic methods in the MetadataUtil class.

Table 3.3 Basic MetadataUtil Methods

Method Name	Description
getRepositories	Gets the ID and name of all repositories registered on the metadata server.
getMetadata	Gets the ID and name of a specified metadata object.
getMetadataSimple	Gets all attributes of a specified metadata object.
getMetadataAllDepths	Gets all of the properties (attributes and associations) of a specified metadata object.
getMetadataObjects	Gets all metadata objects of the requested metadata type.
getMetadataObjectsSubset	Gets a subset of the metadata objects of the requested metadata type.
AddMetadata	Creates a SAS Open Metadata Interface metadata object on the server that has the specified properties.
UpdateMetadata	Updates a SAS Open Metadata Interface metadata object on the server with the specified properties.
DoRequest	Passes an XML-formatted method call to server.

For reference information about each method, see the SAS Java Metadata Interface at support.sas.com/rnd/gendoc/bi/api/.

Using the "Get" Methods

The "get" methods enable you to query metadata repositories:

- Use the getRepositories method to return the repository identifiers necessary to access a SAS metadata repository.
- If you are not sure of which metadata object you need to update, use the getMetadataObjects* methods to list all methods of a specified metadata type. The

getMetadataObjectsSubset method enables you to qualify the objects that are retrieved by passing an XML search string.

- Use the getMetadata* methods to retrieve all or specified sets of properties of a specific object.

Most of these methods require you to pass SAS Open Metadata Interface flags and options to identify the information that you want to retrieve. For example, the GetMetadataObjects* methods support the OMI_XMLSELECT flag and use of an <XMLSelect> option to pass a search string. In addition, the GetMetadata* methods support the OMI_TEMPLATE flag and a <TEMPLATE> option to enable you to identify specific attributes and associations to retrieve in the form of a property string, in addition to other GetMetadata flags.

The SAS Java Metadata Interface Get* methods support all of the flags and options defined for the SAS Open Metadata Interface GetMetadataObjects and GetMetadata methods. For information about these methods, flags, and options, see the "Methods for Reading and Writing Metadata (IOMI Class)" section of the *SAS Open Metadata Interface: Reference*. See specifically "Summary Table of IOMI Flags," "Summary Table of IOMI Options," "Using IOMI Flags", and "Constructing a Metadata Property String."

Usage information about SAS Open Metadata Interface flags and options is provided in the *SAS Open Metadata Interface: User's Guide* in the "Querying All Metadata of a Specified Type" and "Querying Specific Metadata Objects" sections. For information about writing a search string, see "Filtering a GetMetadataObjects Request." For information about creating templates, see "Using Templates." The interface also supports flags that enable you to query multiple repositories at once. This type of query is referred to as a federated query. For information about issuing federated queries, see "Retrieving Objects Across Multiple Repositories."

Using the AddMetadata and UpdateMetadata Methods

The AddMetadata and UpdateMetadata methods are wrappers for the SAS Open Metadata Interface AddMetadata and UpdateMetadata methods. These methods accept an XML-formatted metadata property string as input. See "Constructing a Metadata Property String" in the "Methods for Reading and Writing Metadata (IOMI Class)" section of the *SAS Open Metadata Interface: Reference* for information about how to write a metadata property string.

DoRequest Method

The DoRequest method is a wrapper method for the SAS Open Metadata Interface DoRequest method. The DoRequest method accepts an XML-formatted method call as input. Instructions for formatting an XML method call are provided in "DoRequest" in the "Methods for Reading and Writing Metadata (IOMI Class)" section of the *SAS Open Metadata Interface: Reference*.

Working with the AssociationList Class

The AssociationList class implements the Java List interface to manage the associations between metadata objects. A SAS Open Metadata Interface metadata object instance is defined by its properties: a set of attributes describe the characteristics of the object instance, and a set of associations describe its relationships with other object instances. Native objects representing an instance's attributes are created by using

MdObjectFactory methods. AssociationList objects are created to manage an instance's associations. An AssociationList object is required to represent each association name.

An AssociationList object is created and associated with a native object in one of two ways. The first way is to create the AssociationList object and then call the set "AssociationName" method on the native object. For example, if you want to add a column named "ColumnObject" to a table object named "TableObject" then you would submit the following statements:

```
AssociationList Columns = new AssociationList(TableObject.ASSOCIATION_COLUMNS_NAME");
Columns.add(ColumnObject);
TableObject.setColumns(Columns);
```

In the preceding example,

- the first statement creates an AssociationList object for the Columns association and names it "Columns". ASSOCIATION_*ASSOCIATIONNAME*_NAME is a static variable representing the valid association (or attribute) for an object.
- the second statement adds an object named "ColumnObject" to the Columns AssociationList.
- the third statement associates the Columns AssociationList with object "TableObject" using the set *AssociationName* method.

The second way an AssociationList object is created is by submitting a get *AssociationName* method on the native object. This is the recommended approach for creating an AssociationList object. The following is an example of a get *AssociationName* method call:

```
AssociationList Columns = TableObject.getColumns();
Columns.add(ColumnObject);
```

In the preceding example,

- the first statement specifies to get a list from the metadata server for object "TableObject" all of the objects that have the Columns association name. If a Columns association does not exist, then an AssociationList object is created on the client anyway.
- the second statement adds object "ColumnObject" to the Columns AssociationList that was retrieved or created.

Whenever you create an AssociationList object, the SAS Java Metadata Interface automatically creates an AssociationList object representing the inverse association. For example, for each column listed by getColumns above, the SAS Java Metadata Interface creates AssociationList objects in the object store representing the inverse relationship and persists them to the server along with user-created objects. So, for the preceding example, the ColumnObject.setTable(TableObject) would be explicitly done for the user by the SAS Java Metadata Interface.

The default behavior when an AssociationList object is persisted to the metadata server is to append or modify the associations in the AssociationList object to the appropriate association list on the server. You can specify to replace the entire AssociationList for a given association with a new one by passing the AL_UPDATEALL flag with the set*AssociationName* method. This is done by setting a State on the AssociationList object as follows:

```
TableObject.getColumns().setState(AssociationList.AL_UPDATEALL);
```

Working with the MdObjectStore Class

All Java objects that you create to add or update metadata on the metadata server must be persisted to the server in an MdObjectStore object. The MdObjectStore object serves as a working container for objects that need to be persisted to the metadata server as a group. The object store automatically maintains lists of new, updated, and deleted metadata objects. These lists are used to persist updates to the server and are also used by the event handling interface to track changes among object stores.

An object store is created with the statement:

```
MdObjectStore store = MdObjectFactory.createObjectStore();
```

An object store should be deleted after its changes are persisted to the metadata server. An object store is deleted by using the MdObjectStore dispose method.

Working with the MdServerStore Class

A server store is a special type of object store that is automatically created when you create the object factory. This store contains a placeholder for all objects created or instantiated on the client in any object store, and for all object stores. The store serves as a central event mechanism for the interface: all events from object stores that update or delete objects from the metadata server are sent as events through the server store, and all ID changes are processed in the server store. A client does not need to query the server store when reading or writing metadata; however, the store can be used to verify the existence of an object on the client. You can determine if an object exists in the server store by issuing the following method:

```
serverStore.getObject(myTable.getId());
```

Working with the Util Class

The Util class contains methods used to invoke logging and to control message display within the SAS Java Metadata Interface. Two types of logs can be produced: an XML client/server log and a debug log. The client/server XML log, referred to as 'logging' within the interface documentation, is enabled with MdObjectFactory.getInstance().setLoggingEnabled(true). The destination of the logging output stream is set in Util.setLogStream(). By default, the XML client/server log is directed to standard output.

The debug log (referred to as 'debugging' or 'output') is enabled with the MdObjectFactory.getInstance().setDebug(true). The debug log contains informational messages when an object store is created or deleted and when an association is sent to the metadata server. In addition, it can print the content of the SAS Java Metadata Interface object stores. The destination of the debug output stream is set in Util.setOutputStream.

Other methods in the Util class are used to print information to these logs. The printOutputln() method prints a line of output to the debug log, and the same type of methods are available for the printLoglnClient/Server() methods.

Sample Program

The following is an example of an executable file that contains the code samples described in Chapter 2, "Using the SAS Java Metadata Interface," on page 5. In addition, code is provided that lists the metadata types available on the metadata server, all PhysicalTable objects on the metadata server, and all of the attributes and associations of a specific PhysicalTable object.

```
/**
 * Copyright (c) 2003 by SAS Institute Inc., Cary, NC 27513
 */

package com.sas.metadata;

import com.sas.iom.SAS.*;
import com.sas.meta.*;
import com.sas.meta.SASOMI.*;
import java.beans.*;
import java.rmi.*;
import java.text.*;
import java.util.*;
/**
 * This is a test class that contains the examples for SAS Java Metadata Interface.
 *
 */
public class MdTesterExamples
{

   /**
    * This is the object factory used to create objects.
    */
   private MdObjectFactory mdFact = MdObjectFactory.getInstance();

   /**
    * Default constructor
    */
   public MdTesterExamples()
   {
      mdFact.setDebug(true);
      mdFact.setLoggingEnabled(true);
      if(Example1())
      {
         Util.printOutputln("Connected...");//$NON-NLS-1$
      }else{
         Util.printOutputln("Error Connecting...");//$NON-NLS-1$
      }
      List repositories = Example2();
      Example3();
      Example4((CMetadata)repositories.get(0));
      Example5((CMetadata)repositories.get(0));
      Example6((CMetadata)repositories.get(0));
      List tables = Example7((CMetadata)repositories.get(0));
      Example8(tables);
      Example9(((CMetadata)repositories.get(0)).getId());
```

```
   System.gc();
   System.exit(1);
}


/**
 * This example makes a connection to the metadata server and checks
 * exceptions if there is an error connecting.  The server name, port,
 * user, and password variables must be substituted with actual values.
 * @return True if the connection was successful
 */
public boolean Example1()
{
   String serverName = "MACHINE_NAME";//$NON-NLS-1$
   String serverPort = "9999";//$NON-NLS-1$
   String serverUser = "USERNAME";//$NON-NLS-1$
   String serverPass = "PASSWORD";//$NON-NLS-1$
   MetadataWorkspace workspace = MetadataWorkspace.getWorkspace();

   try
   {
     // This statement makes the connection and sets the handle in the workspace.
      workspace.makeOMRConnection(serverName, serverPort , serverUser, serverPass);

     // The following statements define error handling and error reporting messages.
   }catch (MdException e)
   {
      Throwable t = e.getCause();
      if(t != null)
      {
         String ErrorType = e.getSASMessageSeverity();
         String ErrorMsg = e.getSASMessage();
         if(ErrorType == null)
         {
           // If there is no SAS server message, write a Java/CORBA message.
         }else{
           // If there is a message from the server:
           System.out.println(ErrorType + ": " + ErrorMsg);//$NON-NLS-1$
         }
         if(t instanceof org.omg.CORBA.COMM_FAILURE)
         {
           // If there is an invalid port number or host name:
           System.out.println(e.getLocalizedMessage());
         }else if(t instanceof org.omg.CORBA.NO_PERMISSION)
         {
           // Is there is an invalid user ID or password:
           System.out.println(e.getLocalizedMessage());
         }
      }else{
         // If we cannot find a nested exception, get message and print.
         System.out.println(e.getLocalizedMessage());
      }
      // If there is an error, print the entire stack trace.
      e.printStackTrace();
```

```
      return false;
   }catch (RemoteException e)
   {
      // Unknown exception.
      e.printStackTrace();
      return false;
   }
   // if no errors occur, then a connection is made.
   return true;
}

/**
 * This example retrieves a list of the repositories registered on the server.
 * @return List The list of available repository IDs
 */
public List Example2()
{
   try{
     // Get a list of repositories.
     System.out.println(
          "The repositories contained on this server are: ");//$NON-NLS-1$
     List reposList = MetadataUtil.getRepositories();
     Iterator iter = reposList.iterator();
     while(iter.hasNext())
     {
         CMetadata repository = (CMetadata)iter.next();
         Util.printOutputln("Repository: " + //$NON-NLS-1$
                            repository.getName()
                            + ", "
                            + repository.getFQID());//$NON-NLS-1$
      }
      Util.printOutputln("\n");//$NON-NLS-1$
      return reposList;
   }catch (MdException e)
   {
      e.printStackTrace();
   }
   return new Vector(1);
}

/**
 * This example lists the metadata types available on the metadata server
 * and their descriptions.
 */
public void Example3()
{
 try{
    // Get a list of metadata types available on the server.
    System.out.println("The object types contained on this" +
                       "metadata server are: ");//$NON-NLS-1$
      List Names = new Vector(100);
      List Descs = new Vector(100);
      MetadataUtil.getTypes(Names,Descs);
      Iterator iter2 = Names.iterator();
```

```
      Iterator iter3 = Descs.iterator();
      while(iter2.hasNext() && iter3.hasNext())
      {
         String name = (String)iter2.next();
         String desc = (String)iter3.next();
         System.out.println("Type: " + //$NON-NLS-1$
                            name +
                            ", desc: " + //$NON-NLS-1$
                            desc);
      }
      System.out.println("\n");//$NON-NLS-1$
   }catch (MdException e)
   {
      e.printStackTrace();
   }

}


/**
 * This method creates a table, column, and note on the column, using
 * the store methods. It is a good example of how a wizard-style user interface
 * would utilize the classes would utilize the MdObjectFactory classes.
 *
 * @param Repository CMetadata Object with id of form:  A0000001.A5KHUI98
 */
public void Example4(CMetadata Repository)
{
   if(Repository != null)
   {
      try
      {
         // We have a Repository object.
         // We use the reposFQID method to get its fully qualified ID.
         String reposFQID = Repository.getFQID();

         // We need the short Repository ID to create an object.
         // We use the ReposId method to get the short ID.
         String ReposID = reposFQID.substring(reposFQID.indexOf('.') + 1,
           reposFQID.length());

         // Now we create an object store to hold all our objects.
         // This will be used to maintain a list of objects to persist
         // to the server.
         MdObjectStore myStore = mdFact.createObjectStore();

         // We create a PhysicalTable object named "TableTest".
         PhysicalTable myTable = (PhysicalTable)mdFact.createComplexMetadataObject
            (myStore,
                  null,
                  "TableTest",
                  mdFact.PHYSICALTABLE,
                  ReposID);//$NON-NLS-1$
```

```
            // We create a Column named "ColumnTest".
            Column myColumn = (Column)mdFact.createComplexMetadataObject
               (myStore,
                     null,
                     "ColumnTest",
                     mdFact.COLUMN,
                     ReposID);//$NON-NLS-1$

            // We set attributes of the column.
            myColumn.setColumnName("MyTestColumnName");//$NON-NLS-1$
            myColumn.setSASColumnName("MyTestSASColumnName");//$NON-NLS-1$
            myColumn.setDesc("This is a description of a column");//$NON-NLS-1$

            // We use the get"AssociationName"() method to associate the column
            // with the table. This method creates an AssociationList object for
            // the table object. The inverse association will be created
            // automatically. We could have specified "getColumns(false)" here, but
            // it does not go to the server for temporary objects. If the object
            // already existed, specifying the "false" flag will tell it not to go
            // to the server to get the list of columns. The Add(MetadataObject)
            // method adds myColumn to the AssociationList.
            myTable.getColumns().add(myColumn);

            // We create a note for the column named "NoteTest".
            TextStore myNote = (TextStore)mdFact.createComplexMetadataObject
               (myStore,
                     null,
                     "NoteTest",
                     mdFact.TEXTSTORE,
                     ReposID);//$NON-NLS-1$

            // We use the set"AssociationName" method to associate stored text
            // with the note.
            myNote.setStoredText(
                   "I have some information about the column");//$NON-NLS-1$
            // We associate the note with the column.
            myColumn.getNotes().add(myNote);

            // We pick an object and persist all the new information to the server.
            myTable.updateMetadataAll();

            // Now we clean up the objects, if we are no longer using them.
            myStore.dispose();
         }catch (MdException e)
         {
            e.printStackTrace();
         }
      }
   }


   /**
    * This method reads the newly created objects back from the server.
    * @param repository1 identifies the repository from which to read our objects.
```

```
 */
public void Example5(CMetadata repository1)
{
   if(repository1 != null)
   {
    // First we create an MdObjectStore as a container for all of the objects
    // we will create/read/persist to the server as one collection.
      MdObjectStore myStore = mdFact.createObjectStore();
      try
      {

        // The following statements define GetMetadataObjectsSubset options
        // strings. These XML strings are used in conjunction with SAS Open
        // Metadata Interface flags. The <XMLSELECT> element specifies filter
        // criteria. The <Templates> element specifies the metadata properties
        // to be returned for each object from the server.
        String sOptions = "<XMLSELECT Search=\"@NAME='TableTest'\"/>"+
       "<TEMPLATES><PhysicalTable Id=\"\" Name=\"\" Desc=\"\">"+
       "<Columns/></PhysicalTable>" +
       "<Column Id=\"\" Name=\"\" Desc=\"\"><Notes/></Column>" +
       "<TextStore Id=\"\" Name=\"\" Desc=\"\" StoredText=\"\"/>" +
       "</TEMPLATES>";

        //  The following statements go to the server with a fully-qualified
        //  repository ID, specify the type of object we are searching for
        // (MdObjectFactory.PHYSICALTABLE), and invoke the OMI_XMLSELECT,
        //  OMI_TEMPLATE, and OMI_GET_METADATA flags. OMI_GET_METADATA tells
        //  the server to get all of the attributes specified in the template
        //  for each object that is returned.
         List PhysicalTableList = (MetadataUtil.getMetadataObjectsSubset(myStore,
                                                repository1.getFQID(),
                                                mdFact.PHYSICALTABLE,
                                                MetadataUtil.OMI_XMLSELECT |
                                                MetadataUtil.OMI_TEMPLATE |
                                                MetadataUtil.OMI_GET_METADATA,
                                                sOptions ));
         Iterator iter5 = PhysicalTableList.iterator();
         while(iter5.hasNext())
         {
            PhysicalTable ptable = (PhysicalTable)iter5.next();

          // The preceding method returned a simple form of the requested
          // objects. Simple objects contain only attributes and cannot be persisted
          // to the server. We may want to change data later on so we will create
          // a complex version of the objects. A complex object allows us to edit
          // the object and persist the changes. It also enables us to work with an
          // object's associations.

          PhysicalTable ptable1 =
                  (PhysicalTable)mdFact.createComplexMetadataObject(myStore,
                                                null,
                                                ptable.getName(),
                                                ptable.getCMetadataType(),
                                                ptable.getId(),
```

```
                                                    null);

            // We get a list of columns for this table.
            AssociationList columns = ptable1.getColumns();
            Column column1 = null;
            TextStore columnNote1 = null;
            // Then get individual columns.
            for(int i=0; i < columns.size(); i++)
            {
               Column columnTest = (Column)columns.get(i);
               column1 = (Column)mdFact.createComplexMetadataObject(myStore,
                                                    null,
                                                    columnTest.getName(),
                                                    columnTest.getCMetadataType(),
                                                    columnTest.getId(),
                                                    null);
               if(column1 != null)
               {
                  // We now have a column, and request to get its notes.
                  AssociationList columnNotes = column1.getNotes();
                  for(int j=0; j < columnNotes.size(); j++)
                  {
                     if(columnNotes.size() > 0)
                     {
                        TextStore noteTest = (TextStore)columnNotes.get(0);
                        columnNote1 =
                        (TextStore)mdFact.createComplexMetadataObject
                                                   (myStore,
                                                    null ,
                                                    noteTest.getName(),
                                                    noteTest.getCMetadataType(),
                                                    noteTest.getId(),
                                                    null);
                        if(columnNote1 != null)
                        {
                    // We now have a valid note, and request to print its attributes.
                        System.out.println("TextStore Object: " + columnNote1.getName()
                                                    +//$NON-NLS-1$
                                                    ", " +//$NON-NLS-1$
                                                    columnNote1.getFQID() +
                                                    ", " +//$NON-NLS-1$
                                                    columnNote1.getStoredText() );
                        }
                     }
                  }
               }
            }// End for columns.
         }
      }catch (MdException e)
      {
         e.printStackTrace();

      }
```

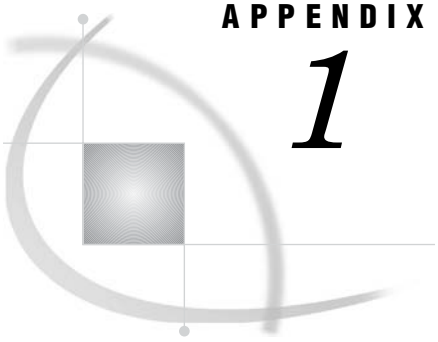
```
   }
}

/**
 * This method deletes the objects we created in repository1.
 * @param repository1
 */
public void Example6(CMetadata repository1)
{
   if(repository1 != null)
   {
      try
      {

         MdObjectStore myStore = mdFact.createObjectStore();
        //  The following statements define GetMetadataObjectsSubset options
        //  strings. These XML strings are used in conjunction with SAS Open
        //  Metadata Interface flags. The <XMLSELECT> element specifies filter
        //  criteria. The <Templates> element specifies the metadata properties
        //  to be returned for each object from the server.

        String sOptions = "<XMLSELECT Search=\"@NAME='TableTest'\"/>"+
                          "<TEMPLATES><PhysicalTable Id=\"\" Name=\"\" "+
                          " Desc=\"\"/></TEMPLATES>";
        // This statement creates a deleteTemplate object.
         String deleteTemplate = "<TEMPLATES><PhysicalTable Id=\"\" Name=\"\">"+
            "<Columns/><Notes/></PhysicalTable>"+
            "<Column><Notes/></Column></TEMPLATES>";

       //  The following statements go to the server with a fully qualified
       //  repository ID, specify the type of object we are searching for
       //  (MdObjectFactory.PHYSICALTABLE),and invoke the OMI_XMLSELECT,
       //  OMI_TEMPLATE, and OMI_GET_METADATA flags. OMI_GET_METADATA tells the
       //  server to get all of the attributes specified in the template for
       //  each object returned.
        List PhysicalTableList = MetadataUtil.getMetadataObjectsSubset(
                                        myStore,
                                        repository1.getFQID(),
                                        mdFact.PHYSICALTABLE,
                                        MetadataUtil.OMI_XMLSELECT |
                                        MetadataUtil.OMI_TEMPLATE |
                                        MetadataUtil.OMI_GET_METADATA,
                                        sOptions );

       // The following statements remove the objects returned by the preceding
       // query from the client and from the server. The code loops through the
       // list of objects and prints the name of each object before deleting it.
       // An event is sent to all object stores to tell them to delete the
       // objects, and to notify their users of a change in the store.

         Iterator iter5 = PhysicalTableList.iterator();
         while(iter5.hasNext())
         {
            PhysicalTable ptable = (PhysicalTable)iter5.next();
```

```
            Set assocNames = ptable.getAssocs().keySet();
            Iterator iter9 = assocNames.iterator();
            System.out.println("PhysicalTable: Associations");//$NON-NLS-1$
            while(iter9.hasNext())
            {
               System.out.println((String)iter9.next());
            }
            mdFact.deleteMetadataObjects(ptable,deleteTemplate);
         }
         myStore.dispose();

      }catch (MdException e)
      {
 e.printStackTrace();
      }
   }
}

/**
 * This method lists the PhysicalTable objects contained in repository
 * "A0000001.A5KE4LY8". The method requests a simple form of the requested
 * objects. If you wish to modify them, you will need to create complex
 * objects that describe the objects.
 *
 * @param mainRepos CMetadata identifies the repository from which to read
 * the objects.
 * @return List containing CMetadata objects
 *
 */
public List Example7( CMetadata mainRepos)
{
   try
   {
      // Print a descriptive message about the request.
      System.out.println(
             "The PhysicalTables contained in repository " + //$NON-NLS-1$
             mainRepos.getName() +
             " are: ");//$NON-NLS-1$

      // We need the short Repository ID to pass in the method.
      // We use the ReposId method to get the short ID.
      String reposID = mainRepos.getFQID();

      // We get a list of "PhysicalTable" objects.
      List
      physicalTables = MetadataUtil.getMetadataObjectsSubset
             (mdFact.getStore(),
             reposID,                          // Repository to Search
             mdFact.PHYSICALTABLE,             // Type to search for
             MetadataUtil.OMI_GET_METADATA |  // Return the object
             MetadataUtil.OMI_ALL_SIMPLE ,    // Return its simple attributes
             "" );//$NON-NLS-1$

      // We print information about them.
```

```
      Iterator iter4 = physicalTables.iterator();
      while(iter4.hasNext())
      {
         PhysicalTable ptable = (PhysicalTable)iter4.next();
         System.out.println("PhysicalTable: " + //$NON-NLS-1$
                            ptable.getName() +
                            ", " + //$NON-NLS-1$
                            ptable.getFQID() +
                            ", " + //$NON-NLS-1$
                            ptable.getDesc() );
      }
      System.out.println("\n");//$NON-NLS-1$

      // We return the list so that it can be used.
      return physicalTables;
   }catch (MdException e)
   {
      e.printStackTrace();
   }

   // If the method fails, then execute this:
   return new Vector(1);
}


/**
 * This method gets all the information for a specific PhysicalTable object.
 *
 *
 * @param physicalTables
 */
public void Example8(List physicalTables)
{
   if(physicalTables.size() == 0)
   {
      return;
   }
   MdObjectStore myStore = mdFact.createObjectStore();
   try
   {

   // First we will print a message describing out intentions.
   System.out.println("Get the first PhysicalTable object found and list
      all of its properties.");
   PhysicalTable testTable = (PhysicalTable)physicalTables.get(0);
   testTable = (PhysicalTable)mdFact.createComplexMetadataObject(myStore,
                                  null,
                                  testTable.getName(),
                                  testTable.getCMetadataType(),
                                  testTable.getFQID(),
                                  null);

   // We now build a list of all the complex attributes.
   List complex = new Vector(10);
```

```
        Iterator iter = testTable.getAssocs().keySet().iterator();
        while (iter.hasNext())
        {
          complex.add((String)iter.next());
        }

       // We specify templates to get notes and documents defined for our columns.
       String template ="<Templates><Column><Notes/><Documents/></Column>" ;

       // We get the information for our testTable.
       testTable = (PhysicalTable)MetadataUtil.getMetadataAllDepths
                      (testTable,                    // Object to get info for
                       null,                         // Simple attributes to get
                       complex,                      // Associations to get
                       template,                     // Template for returned objects
                       MetadataUtil.OMI_ALL |        // Get all attributes and
                                                     // associations for returned
                                                     // objects
                       MetadataUtil.OMI_ALL_SIMPLE); // Get all attributes for
                                                     // returned objects

       System.out.println("PhysicalTable: " +
                          testTable.getName() +
                          ", " +
                          testTable.getFQID() +
                          ", " +
                          testTable.getDesc());
       System.out.println("The columns in this table are: ");

       // We then list the columns on the table. Specifying 'false' causes a Columns-Table
       // association to be created if one does not already exist; if the association
       // already exists, the software returns the current list on the client.

       AssociationList columns = testTable.getColumns(false);
       for(int i = 0; i < columns.size(); i++)
       {
          Column column = (Column)columns.get(i);
          System.out.println("\tColumn: " + column.getName() + "," +
                             column.getFQID());  //$NON-NLS-1$
       }
       System.out.println("\n");

     }catch (MdException e)
      {
         e.printStackTrace();
      }
      myStore.dispose();
    }

   /**
    * This example gets DeployedComponent objects.
    * @param reposID
    */
    public void Example9(String reposID)
```

```
 {
   try
   {
      // We use the <XMLSELECT> option to do an explicit search and use
      // templates to specify the attributes we want to retrieve.
      System.out.println("Do we have a DeployedComponent in this repository that
        has a classidentifier specified?");
      System.out.println("Using Template: ");
      String sOptions =
"<XMLSELECT Search=\"@CLASSIDENTIFIER='440196D4-90F0-11D0-9F41-00A024BB830C'\"/>" +
"<TEMPLATES><DeployedComponent Id=\"\" Name=\"\" ClassIdentifier=\"\">"+
"<SourceConnections/></DeployedComponent>"+
"<TCPIPConnection Id=\"\" HostName=\"\" Port=\"\" CommunicationProtocol=\"\" " +
"ApplicationProtocol=\"\"/></TEMPLATES>";

      System.out.println(MetadataUtil.formatXML(sOptions));
      List DeployedComponentList =(MetadataUtil.getMetadataObjectsSubset(
                                          mdFact.getStore(),
                                          reposID,
                                          mdFact.DEPLOYEDCOMPONENT,
                                          MetadataUtil.OMI_XMLSELECT |
                                          MetadataUtil.OMI_TEMPLATE |
                                          MetadataUtil.OMI_GET_METADATA,
                                          sOptions ));
      Iterator iter5 = DeployedComponentList.iterator();
      while(iter5.hasNext())
      {
         DeployedComponent dcomp = (DeployedComponent)iter5.next();
         System.out.println("DeployedComponent: " +
                            dcomp.getName() +
                            ", " +
                            dcomp.getFQID() +
                            ", " +
                            dcomp.getDesc() );
      }
      System.out.println("\n");
   }catch (MdException e)
   {
      e.printStackTrace();
   }
}

  /**
   * The main method for the class
   */
  public static void main(String[] args)
  {
     new MdTesterExamples();
  }

}
```



APPENDIX

1

Recommended Reading

Recommended Reading

Here is the recommended reading list for this title:

- *SAS Open Metadata Interface: Reference*

For a complete list of SAS publications, see the current *SAS Publishing Catalog*. To order the most current publications or to receive a free copy of the catalog, contact a SAS representative at

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513
Telephone: (800) 727-3228*
Fax: (919) 677-8166
E-mail: **sasbook@sas.com**
Web address: **support.sas.com/pubs**
* For other SAS Institute business, call (919) 677-8000.

Customers outside the United States should contact their local SAS office.

Your Turn

If you have comments or suggestions about *SAS® 9.1.3 Java Metadata Interface: User's Interface* , please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
E-mail: **yourturn@sas.com**

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
E-mail: **suggest@sas.com**