THE POWER TO KNOW.

SAS® 9.3 Integration Technologies

Java Client Developer's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc 2011. *SAS® 9.3 Integration Technologies: Java Client Developer's Guide*. Cary, NC: SAS Institute Inc.

SAS® 9.3 Integration Technologies: Java Client Developer's Guide

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

1st electronic book, July 2011

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/publishing** or call 1-800-727-3228.

Contents

What's New in Integration Technologies Java Client Development

Overview

SAS 9.3 Integration Technologies provides a new zero-configuration workspace server interface. In addition, the Logging service is deprecated.

Zero-Configuration Workspace Servers

The new zero-configuration workspace server interface enables you to launch a workspace server without specifying any connection information or configuring server metadata. Zero-configuration workspace servers require that SAS Foundation is installed on the same Windows machine where your client is running.

Logging Service Deprecation

The Logging service is deprecated in SAS 9.3. Custom SAS clients should use Log4j to perform logging tasks.

Chapter 1

Concepts

Developing Java Clients

The application programming interfaces that are provided with SAS Integration Technologies enable you to develop Java-based distributed applications that are integrated with the SAS platform. SAS Integration Technologies includes the following features for developing Java clients:

- The Connection Factory interface, which enables Java programs to communicate with Integrated Object Model (IOM) servers through an IOM Bridge connection. The connection factory enables you to obtain server attributes from a SAS Metadata Server, from the Information Service (which is part of SAS Foundation Services), or directly from an application program. When it is used with the SAS Open Metadata Architecture, the Java Connection Factory interface provides the following:
  - connections to new types of IOM servers (SAS Metadata Servers, SAS Stored Process Servers, and SAS OLAP Servers) in addition to SAS Workspace Servers
  - the ability to use load balancing for workspace and stored process servers and spawners
- SAS Foundation Services, which extends Java application development beyond access to IOM servers. The following core infrastructure services are provided:
  - client connections to application servers (including the Java Connection Factory interface previously mentioned)
  - dynamic service discovery
  - user authentication and profile management
  - session context management

  - metadata and content repository access
  - activity logging

  SAS Foundation Services also includes extension services for event management, information publishing, and stored process execution.
- The SAS Foundation Services Facade, which includes convenience services that Web application developers can use to easily access the most commonly used SAS Foundation Services methods and objects.

Use of this software requires some knowledge of distributed programming. However, the software rigorously conforms to Java distributed programming standards such as CORBA and JDBC. Whether you are developing an applet, a stand-alone application, a servlet, or an enterprise JavaBean, you can focus your attention on exploiting the features of the SAS platform rather than determining how to communicate with it.

SAS Integration Technologies supports any Java integrated development environment (IDE), including Eclipse, JBuilder, and SAS AppDev Studio.

Java Client Installation and JRE Requirements

Client Installation

To install the Java client software, you must install SAS Foundation Services.

JRE Requirements

The current release of the Java client software requires the Java Runtime Environment (JRE), Version 5.

To compile and run the code examples included in the Java client development documentation, you must include Java archives in your classpath as specified in the following table.

Table 1.1 *Requirements for Code Samples*

Samples	Java Archives Needed
All samples	`sas.core.jar`
	`sas.svc.connection.jar`
SAS Metadata Server samples	`sas.oma.joma.jar`
	`sas.oma.joma.rmt.jar`
	`sas.oma.omi.jar`
	`sas.oma.util.jar`
	`sas.svc.connection.platform.jar`

To run the IDL-to-Java compiler or run the binder utility, you must include **`sas.iom.tools.jar`** in your classpath.

For help setting your classpath, see the documentation for your Java Runtime Environment.

Java Client Security

For an overview and understanding of security for the SAS Open Metadata Architecture, see the *SAS Intelligence Platform: Security Administration Guide*.

The IOM Bridge for Java has the ability to encrypt all messages exchanged with the IOM server by using a two-tiered security solution. The first tier is a SAS proprietary encryption algorithm. The second tier contains standards-based RC2, RC4, DES, and Triple DES encryption algorithms.

The SAS proprietary encryption algorithm (SASPROPRIETARY) is appropriate to use when you want to prevent accidental exposure of information while it is being transmitted over a network between an IOM Bridge for Java and an IOM server. Access to this encryption algorithm is included with your Base SAS license, and the Java implementation is integrated into the IOM Bridge for Java.

The second-tier encryption algorithms are appropriate to use when you want to prevent exposure of secret information. In other words, using these algorithms makes it extremely difficult to discover the content of messages exchanged between an IOM Bridge for Java and an IOM server. To use these algorithms you must license SAS/SECURE software.

In addition to encryption, SAS/SECURE software also supports message authentication codes (MAC). A MAC is a few bytes of information that is appended to a message to allow the receiver to confirm that the message has not been altered in transit.

Instructions for the security features of the IOM Bridge for Java are included with the documentation for the **`com.sas.services.connection`** class. Those instructions contain some tips on how to configure the IOM server, but more complete information is available in the documentation for Base SAS software. Installation instructions and usage information for second-tier encryption algorithms is provided in the documentation for SAS/SECURE software.

Using the IOM Server

Overview of Using the IOM Server

This section introduces the steps necessary to construct and execute a Java application that uses the IOM server. As you become more familiar with Java client programming for the IOM server, you can build on these steps to exploit the more sophisticated features of the IOM server.

- a SAS Metadata Server.
- server parameters supplied directly in the source code. (You can supply a **`ManualConnectionFactoryConfiguration`** object directly in the source code. For details, see “Connecting with Directly Supplied Server Attributes ” on page 8).

The Connection Service can connect to SAS Workspace Servers, other metadata servers, SAS OLAP Servers, and SAS Stored Process Servers.

Using a Metadata Server with the Connection Service

If you are using a metadata server, the first step in developing and running a client program is to make sure you have access to a properly configured server. You can access a server by reading the connection information from a SAS Metadata Server.

As is the case in client development, you can start with a basic server configuration and then move into more a sophisticated configuration over time.

After the IOM server has been configured, you can begin developing a Java client for the IOM server.

Connecting a Java Client to an IOM Server

Java clients can use the Java Connection Factory interface to access an IOM server by performing the following steps:

1. From the Java Connection Factory, obtain a connection to an IOM server. Then, obtain the remote object reference connected to that IOM server and narrow it to the appropriate remote interface.
2. Use Java CORBA stubs for IOM objects and JDBC connection objects to exploit the power of SAS in the IOM server.
3. Return the connection to the Java Connection Factory for disconnection or reuse.

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 and learn about Java client programming for the IOM server in greater detail.

Chapter 2

Using the Java Connection Factory

Using the Java Connection Factory

Overview of the Java Connection Factory

The Java Connection Factory interface of the Connection Service provides the following features:

- IOM Bridge connections to IOM servers
- scalability through pooling and server failover
- support for load-balancing spawners

Configuring the Java Connection Factory and obtaining a connection are the first steps in using an IOM server. To connect to an IOM server, you can use methods in the classes that implement the **`ConnectionFactoryInterface`** interface.

Supplying Connection Information

In a Java client program, there are several ways to supply the Java Connection Factory with the information that it needs in order to connect to an IOM server:

- You can place the required information directly in the client program. For details, see "Connecting with Directly Supplied Server Attributes " on page 8 . Connections can be made one at a time on an as-needed basis, or you can set up a pool of connections (see "Pooling with Directly Supplied Server Attributes " on page 19) to be shared and reused across multiple Java client applications and multiple connection requests. Connection pooling is secure, and it can dramatically reduce connection times in environments where one or more client applications make frequent but brief requests for IOM services.
- Alternatively, you can obtain the required information from a managed, secure SAS Metadata Server using indirect logical names. The Java Connection Factory supports metadata access from a SAS Metadata Server. For details, see "Connecting with Server Attributes Read from a SAS Metadata Server " on page 10 . When you use this method, the metadata server administrator decides whether to use connection pooling. (See "Pooling with Server Attributes Read from a Metadata Server" on page 22 .)
- If you configure SAS Foundation Services, then you can obtain the required information from a SAS Metadata Server by using the Information Service. For details, see "Connecting with Server Attributes Read from the Information Service " on page 12 . When you use this method, the metadata server administrator decides whether to use connection pooling. (See "Pooling with Server Attributes Read from a Metadata Server" on page 22 .)
- In Windows environments, you can create a connection to a local server without specifying connection information. For details, see "Connecting to a Zero-Configuration Workspace Server" on page 14.

Using Connection Factory Configurations, Connection Factories, and Connections

To create a connection to an IOM server, perform the following steps:

1. Create the connection factory configuration. You must configure a connection factory to identify the location and type of IOM server to which you want to connect. For example, to create a connection to host foo.bar.abc.com at port 1234, use the following code:

```
String classID = Server.CLSID_SAS;
String host = "foo.bar.abc.com";
int port = 1234;
Server server = new BridgeServer(classID,host,port);
ConnectionFactoryConfiguration cxfConfig =
   new ManualConnectionFactoryConfiguration(server);
```

2. Create the connection factory. After creating a connection factory configuration, you must find or create a connection factory that matches the configuration. The connection factory manager maintains a set of connection factories, and, if one of these connection factories matches your configuration, that factory is returned. Otherwise, the connection factory manager creates a new connection factory and returns it. For example, to create a connection factory that matches the connection factory configuration in step 1, use the following code:

```
ConnectionFactoryConfiguration cxfConfig = ...
ConnectionFactoryManager cxfManager =
   new ConnectionFactoryManager();
ConnectionFactoryInterface cxf =
   cxfManager.getFactory(cxfConfig);
```

3. Create the connection. To obtain a connection to the IOM server, you must provide a user name and a password that are valid on the server. For example, to get a connection from the connection factory that you created in step 2, use the following code:

```
ConnectionFactoryInterface cxf= ...
String userName = "myName";
String password = "mySecret";
ConnectionInterface cx =
   cxf.getConnection(userName,password);
```

4. Narrow the connection. When a connection factory returns a connection, the connection is a generic interface for communicating with remote objects on the server. You can convert the generic interface to a server-specific interface through a mechanism called narrowing. Narrowing is equivalent to the casting mechanism that is used with remote object references. The connection factory contains classes that are necessary to narrow a generic interface reference to a workspace server reference. To narrow the connection that is obtained in step 3, use the following code:

```
ConnectionInterface cx = ...
org.omg.CORBA.Object obj = cx.getObject();
com.sas.iom.SAS.IWorkspace iWorkspace =
   com.sas.iom.SAS.IWorkspaceHelper.narrow(obj);
```

5. End the connection. After you are finished using a connection that you have obtained from the Java Connection Factory, you must return it to the factory by calling the **`close()`** method on the connection. For details, see "Closing a Connection to the Java Connection Factory" on page 24 . This process is the same whether you are using connection pooling or making single connections. It is also the same whether you provide information about the IOM servers directly in your client program or indirectly using a metadata server.

6. Shut down the connection factory. When you are finished with the instance of the Java Connection Factory itself and you no longer need to request connections from it, you must shut it down by calling the **`shutdown()`** method or the **`destroy()`** method. For details, see "Shutting Down the Java Connection Factory" on page 24.

Connection Factory Logging

The Java Connection Factory logs diagnostic and status messages and writes them to output for use in debugging or performance monitoring. For details, see "Logging Java Connection Factory Activity " on page 16.

Connecting with Directly Supplied Server Attributes

Overview of Connecting with Directly Supplied Server Attributes

In order to make a connection to an IOM server, you must give the Java Connection Factory specific information about the server and about the desired connection. The quickest and simplest method of providing this information is to place it directly into the client program when you create the **`BridgeServer`** object. The following attributes can be provided:

host
: specifies the IP address of the machine where the IOM server or object spawner is running. This attribute is required.

port
: specifies the TCP port that the IOM server or object spawner is listening on for connections. This attribute is required.

encryptionPolicy
: specifies whether IOM Bridge for Java should attempt to negotiate with the server over which encryption algorithm to use and what to do if the negotiations fail. This attribute is optional. Possible values are as follows:

  none
  : specifies not to use encryption. This is the default.

  optional
  : specifies to attempt to use encryption but, if algorithm negotiation fails, continue with an unencrypted connection.

  required
  : specifies to attempt to use encryption but, if algorithm negotiation fails, fail the connection.

encryptionAlgorithms
: specifies the list of algorithms that you are willing to use in order of preference. Values in the list should be separated by commas and chosen from SASPROPRIETARY, RC2, RC4, DES, or TRIPLEDES. This attribute is optional. If no value is specified, then one of the server's preferred algorithms is used. It is ignored entirely if the value for encryptionPolicy is *none*.

  Note: If you do not have a license for SAS/SECURE software, then only the SASPROPRIETARY algorithm is available.

encryptionContent
specifies which messages should be encrypted if encryption is used. This attribute is optional, and it is ignored entirely if the value for encryptionPolicy is *none*. Possible values are as follows:

all
encrypts all messages. This is the default.

authentication
encrypts only messages that contain user name and password information.

Example of Connecting with Directly Supplied Server Attributes

The Java code in this example demonstrates how to create a BridgeServer object to provide information to the Java Connection Factory and obtain a connection. For an example showing how to use a connection, see "Java Connection Factory Language Service Example" on page 15.

The last two statements in this example show how to dispose of a connection. For details about this procedure, see "Returning Connections to the Java Connection Factory " on page 24.

```
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SAS.IWorkspaceHelper;
import com.sas.services.connection.BridgeServer;
import com.sas.services.connection.ConnectionFactoryAdminInterface;
import com.sas.services.connection.ConnectionFactoryConfiguration;
import com.sas.services.connection.ConnectionFactoryInterface;
import com.sas.services.connection.ConnectionFactoryManager;
import com.sas.services.connection.ConnectionInterface;
import com.sas.services.connection.ManualConnectionFactoryConfiguration;
import com.sas.services.connection.Server;

// identify the IOM server
String classID = Server.CLSID_SAS;
String host = "rnd.fyi.sas.com";
int port = 5310;
Server server = new BridgeServer(classID,host,port);

// make a connection factory configuration with the server
ConnectionFactoryConfiguration cxfConfig =
   new ManualConnectionFactoryConfiguration(server);

// get a connection factory manager
ConnectionFactoryManager cxfManager = new ConnectionFactoryManager();

// get a connection factory that matches the configuration
ConnectionFactoryInterface cxf = cxfManager.getFactory(cxfConfig);

// get the administrator interface
ConnectionFactoryAdminInterface admin = cxf.getAdminInterface();

// get a connection
String userName = "abcserv";
String password = "abcpass";
ConnectionInterface cx = cxf.getConnection(userName,password);
org.omg.CORBA.Object obj = cx.getObject();
```

```
IWorkspace iWorkspace = IWorkspaceHelper.narrow(obj);

< insert iWorkspace workspace usage code here >

cx.close();
// tell the factory that it can destroy unused connections
admin.shutdown();
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Connecting with Server Attributes Read from a SAS Metadata Server

Overview of Connecting with Server Attributes from Metadata

The Java Connection Factory enables you to obtain the server connection information from a SAS Metadata Server by using indirect logical server names. The main advantage of this method is that you can maintain and update the IOM server and connection information without changing your client programs. This method also provides additional security features if you are using connection pooling.

To use this method, you must provide the client program with instructions for connecting to the metadata server, the name of the information that you want to search for, and the repository within the metadata server for performing the search. To connect to the metadata server, you must first create an instance of **`BridgeServer`** that contains the appropriate attributes for the metadata server. For a complete list of the attributes that you can provide, see the documentation for the **`BridgeServer`** class.

Note: You can read the metadata by either connecting to the metadata server directly or by using the Information Service. For details about using the Information Service, see “Connecting with Server Attributes Read from the Information Service ” on page 12.

Example of Connecting with Server Attributes from Metadata

The following example code shows how to initialize and use the Java Connection Factory with information from a SAS Metadata Server directory. For information about how to use the object reference, see “Java Connection Factory Language Service Example” on page 15.

The last three statements in the example code show how to dispose of object references. For details about this procedure, see “Returning Connections to the Java Connection Factory ” on page 24.

```
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SAS.IWorkspaceHelper;
import com.sas.meta.SASOMI.IOMI;
import com.sas.meta.SASOMI.IOMIHelper;
import com.sas.services.connection.BridgeServer;
import com.sas.services.connection.ConnectionFactoryAdminInterface;
import com.sas.services.connection.ConnectionFactoryConfiguration;
import com.sas.services.connection.ConnectionFactoryInterface;
```

```
import com.sas.services.connection.ConnectionFactoryManager;
import com.sas.services.connection.ConnectionInterface;
import com.sas.services.connection.ManualConnectionFactoryConfiguration;
import com.sas.services.connection.omr.OMRConnectionFactoryConfiguration;
import com.sas.services.connection.Server;
String classID = Server.CLSID_SASOMI;
String host = "omr.pc.abc.com";
int port = 8561;

// Set the credentials for the metadata server connection. If connecting to
// a pooled server, these should be the credentials for the pooling
// administrator.
String userName_omr = "Adm1";
String password_omr = "Adm1pass";

// Step 1. Create a connection factory configuration for the metadata server
// and get a connection factory manager.
Server omrServer = new BridgeServer(classID,host,port);
ConnectionFactoryConfiguration cxfConfig_omr =
   new ManualConnectionFactoryConfiguration(omrServer);
ConnectionFactoryManager cxfManager = new ConnectionFactoryManager();

// Step 2. Create a connection factory for the metadata server connection
// factory configuration.
ConnectionFactoryInterface cxf_omr = cxfManager.getFactory(cxfConfig_omr);

// Step 3. Get a connection to the metadata server.
ConnectionInterface cx_omr = cxf_omr.getConnection(userName_omr,password_omr);

// Step 4. Narrow the connection from the metadata server.
org.omg.CORBA.Object obj_omr = cx_omr.getObject();
IOMI iOMI = IOMIHelper.narrow(obj_omr);
String reposID = "A0000001.A1234567";
String name= "login015Logical";

// Step 5. Create a connection factory configuration for the server by passing
// the server logical name to the metadata server.
ConnectionFactoryConfiguration cxfConfig =
   new OMRConnectionFactoryConfiguration(iOMI,reposID,name);

// Step 6: Get a connection factory that matches the server's connection
// factory configuration.
ConnectionFactoryInterface cxf = cxfManager.getFactory(cxfConfig);
// Set the credentials for the server connection.
String userName = "citynt\\use1";
String password = "use1pass";
String domain = "citynt";

// Step 7: Get a connection to the server.
ConnectionInterface cx = cxf.getConnection(userName,password,domain);

// Step 8: Narrow the connection from the server.
org.omg.CORBA.Object obj = cx.getObject();
IWorkspace iWorkspace = IWorkspaceHelper.narrow(obj);

< insert iWorkspace workspace usage code here >
```

```
// Step 9: Close the workspace connection and shutdown the connection factory.
cx.close();
cxf.shutdown();

// Step 10: Close the metadata server connection and shutdown the connection
// factory.
cx_omr.close();
cxf_omr.shutdown();
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Connecting with Server Attributes Read from the Information Service

Overview of Connecting with Server Attributes from the Information Service

The Java Connection Factory enables you to obtain server connection information from a metadata server by using the Information Service component of SAS Foundation Services. The Information Service enables you to access multiple SAS Metadata Repositories simultaneously and perform searches across all metadata sources.

Before you use this method, you must do the following:

- set up the User Service. For more information, see "Understanding and Editing the User Service" in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide* and **`com.sas.services.user`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.
- set up the Information Service. For more information, see "Modifying the Information Service Configuration " in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide* and **`com.sas.services.information`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

Note: The ConnectionFactory class that is used in the other connection methods is not integrated with SAS Foundation Services. To use the Information Service to connect, you must use the PlatformConnectionFactory class in **`com.sas.services.connection.platform`**.

Example of Connecting with Server Attributes from the Information Service

The following example code shows how to initialize and use the Java Connection Factory with information from the Information Service. For information about how to use the object reference, see "Java Connection Factory Language Service Example" on page 15.

The last three statements in the example code show how to dispose of object references. For details about this procedure, see "Returning Connections to the Java Connection Factory " on page 24.

```
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SAS.IWorkspaceHelper;
import com.sas.services.connection.ConnectionFactoryConfiguration;
import com.sas.services.connection.ConnectionInterface;
import com.sas.services.connection.platform.PlatformConnectionFactoryInterface;
import
com.sas.services.connection.platform.PlatformConnectionFactoryConfiguration;
import com.sas.services.information.RepositoryInterface;
import com.sas.services.information.metadata.LogicalServerInterface;
import com.sas.services.user.UserContextInterface;
import com.sas.services.user.UserServiceInterface;

 set up the User Service and create a UserServiceInterface uService >

// Step 1. Create a user context using the User Service
UserContextInterface cxfUser = uService.newUser("user1","user1pw",
   "user1domain");

 set up the Information Service and define a repository repos >

// Step 2. Identify the repository
RepositoryInterface cxfRepos cxfUser.getRepository("repos");

// Step 3. Identify the IOM service
LogicalServerInterface logicalServer =
   cxfRepos.fetch("A50IFJQG.AQ000002/LogicalServer");

// Step 4. Create a connection factory configuration
ConnectionFactoryConfiguration cxfConfig = new
   PlatformConnectionFactoryConfiguration(logicalServer);

// Step 5. Get a connection factory manager
PlatformConnectionFactoryManager cxfManager = new
   PlatformConnectionFactoryManager();

// Step 6. Get a connection factory using the configuration
PlatformConnectionFactoryInterface cxf =
   cxfManager.getPlatformFactory(cxfConfig);

// Step 7. Get a connection
ConnectionInterface cx = cxf.getConnection(cxfUser);

// Step 8. Narrow the connection
org.omg.CORBA.Object obj = cx.getObject();
IWorkspace iWorkspace = IWorkspaceHelper.narrow(obj);

< insert iWorkspace workspace usage code here >

// Step 9. Close the connection when finished
cx.close();
cxf.getAdminInterface().destroy();
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Connecting to a Zero-Configuration Workspace Server

Overview of Connecting to a Zero-Configuration Workspace Server

Zero-configuration workspace server connections enable you to create a local workspace server without specifying any connection attributes. This feature is specific to Windows environments.

Zero-configuration workspace server connections do not require an object spawner or a SAS metadata server.

To create a zero-configuration workspace server connection, you must meet the following requirements:

- SAS 9.3 or later is installed on the same machine where the client is running.
- Your client and your SAS software run on a Windows machine.
- The sspiauth.dll file is in either your system PATH, your java.library.path, or in the home directory of your Java client.

A zero-configuration workspace server runs as a child process of the Java process that creates it. The server runs as the same user ID as the owner of the Java process.

The **`com.sas.services.connection.ZeroConfigWorkspaceServer`** class creates the zero-configuration connection.

Example of Connecting to a Zero-Configuration Workspace Server

The Java code in this example demonstrates how to create a Server

```
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SAS.IWorkspaceHelper;
import com.sas.services.connection.ConnectionFactoryManager;
import com.sas.services.connection.ManualConnectionFactoryConfiguration;
import com.sas.services.connection.SecurityPackageCredential;
import com.sas.services.connection.ZeroConfigWorkspaceServer;


 server = new ZeroConfigWorkspaceServer();
 config = new ManualConnectionFactoryConfiguration(server);
 manager = new ConnectionFactoryManager();
 factory = manager.getFactory(config);
 cred = new SecurityPackageCredential();
 cx = factory.getConnection(cred);
 try {

   //  Narrow the connection from the server.
   org.omg.CORBA.Object obj = cx.getObject();
   IWorkspace iWorkspace = IWorkspaceHelper.narrow(obj);
```

```
< insert iWorkspace workspace usage code here >

 }
 finally {cx.close();}
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Java Connection Factory Language Service Example

The SAS language component of the IOM server enables you to submit SAS code for processing and to obtain output and information in the SAS log. The following example shows you how to do this and also shows you how to use CORBA holder classes to handle output parameters.

The following example assumes that you already have a reference (see "Using the Java Connection Factory" on page 6) to a workspace object.

```
import com.sas.iom.SAS.ILanguageService;
import com.sas.iom.SAS.ILanguageServicePackage.CarriageControlSeqHolder;
import com.sas.iom.SAS.ILanguageServicePackage.LineTypeSeqHolder;
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SASIOMDefs.StringSeqHolder;

//use the Connection Factory to get a reference to a workspace object stub
//IWorkspace iWorkspace = ...
ILanguageService sasLanguage = iWorkspace.LanguageService();
sasLanguage.Submit(data a;x=1;run;proc print;run;);
CarriageControlSeqHolder logCarriageControlHldr =
   new CarriageControlSeqHolder();
LineTypeSeqHolder logLineTypeHldr = new LineTypeSeqHolder();
StringSeqHolder logHldr = new StringSeqHolder();
sasLanguage.FlushLogLines(Integer.MAX_VALUE,logCarriageControlHldr,
   logLineTypeHldr,logHldr);
String[] logLines = logHldr.value;
CarriageControlSeqHolder listCarriageControlHldr =
   new CarriageControlSeqHolder();
LineTypeSeqHolder listLineTypeHldr = new LineTypeSeqHolder();
StringSeqHolder listHldr = new StringSeqHolder();
sasLanguage.FlushListLines(Integer.MAX_VALUE,listCarriageControlHldr,
listLineTypeHldr,listHldr);
String[] listLines = listHldr.value;
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Logging Java Connection Factory Activity

In SAS 9.3 and later, the connection factory uses Log4j to perform logging tasks. For information about Log4j, see **http://logging.apache.org/log4j/1.2/manual.html**.

To configure Log4j for your client, you should create a Log4j configuration file. The configuration file can be either an XML file or a properties file. You can specify the location of your configuration file by using the **-Dlog4j.configuration=*path-to-file*** on your JVM command line.

Your Log4j configuration file defines the loggers and appenders for your SAS classes. For details about creating your configuration file, see **http://logging.apache.org/log4j/1.2/manual.html**.

For example, the following properties file specifies a root logger for all logging activity, a logger for all classes within the **com.sas** hierarchy, and a logger for **com.sas.services.connection**:

```
# http://logging.apache.org/log4j/docs/manual.html
# Available log priority levels are: TRACE, DEBUG, INFO, WARN, ERROR, FATAL
log4j.rootCategory=ERROR, A1
log4j.logger.com.sas=WARN, SASConsoleAppender
log4j.additivity.com.sas=false
log4j.logger.com.sas.services.connection=INFO, SASConsoleAppender
log4j.additivity.com.sas.services.connection=false
#
# A1 is a ConsoleAppender
# note: when the SAS foundation Logging Service reconfigures the root logger
#       this appender will be closed.
#
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-5p [%t] - %m%n
#
# SASConsoleAppender is a ConsoleAppender
#
log4j.appender.SASConsoleAppender=org.apache.log4j.ConsoleAppender
log4j.appender.SASConsoleAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.SASConsoleAppender.layout.ConversionPattern=%-5p [%t] - %m%n
```

Using Failover

Failover enables a Java Connection Factory to redirect connection requests in the event of server unavailability.

Connection factories that are configured to use failover provide enhanced reliability by using a group of redundant servers called a failover cluster rather than single server. If a server in the failover cluster is unavailable, then the connection factory redirects connection requests to the next server in the failover cluster.

The following code fragment configures a connection factory to use failover:

```
String classID = Server.CLSID_SAS;
Server server0 = new BridgeServer(classID,"foo0.bar.abc.com",1234);
Server server1 = new BridgeServer(classID,"foo1.bar.abc.com",1234);
Server[] servers = {server0,server1};
Cluster cluster = new FailoverCluster(servers);
ConnectionFactoryConfiguration cxfConfig =
   new ManualConnectionFactoryConfiguration(cluster);
```

Note: The connection factory uses the servers in a failover cluster in the order in which they are specified.

Using Load Balancing

Load balancing enables a Java Connection Factory to distribute server load between a cluster of redundant servers. For more information about load balancing, see "Understanding Server Load Balancing" in *SAS Intelligence Platform: Application Server Administration Guide*.

Note: Load balancing can be used with SAS Stored Process Servers and SAS Workspace Servers only .

A connection factory that is configured for load balancing uses a group of redundant servers called a load-balancing cluster to send connection requests to the server that has the least load. If a server in the load-balancing cluster is unavailable, then connection requests are sent to other servers instead.

The following code fragment configures a connection factory to use load balancing:

```
String classID = Server.CLSID_SAS;
Server server0 = new BridgeServer(classID,foo0.bar.abc.com,1234);
Server server1 = new BridgeServer(classID,foo1.bar.abc.com,1234);
Server[] servers = {server0,server1};
Cluster cluster = new LoadBalancingCluster(servers);
ConnectionFactoryConfiguration cxfConfig =
   new ManualConnectionFactoryConfiguration(cluster);
```

Note: Load-balancing clusters require additional configuration on the server side. For details, see "Understanding Server Load Balancing" in *SAS Intelligence Platform: Application Server Administration Guide*.

Using Connection Pooling

Overview of Pooling

Pooling enables you to create a pool of connections to IOM servers. These connections are then shared and reused among multiple client applications. Pooling improves the efficiency of connections between clients and servers because clients use the connections only when they need to process a transaction.

Note: Pooling can be used with SAS Workspace Servers only.

Pooling is most useful for applications that require the use of an IOM server for a short period of time. Because pooling reduces the wait that an application incurs when

establishing a connection to SAS, pooling can reduce connection times in environments where one or more client applications make frequent but brief requests for IOM services. For example, pooling is useful for Web applications, such as JavaServer Pages (JSPs).

Pooling is least useful for applications that acquire an IOM server and use the server for a long period of time. A pooled connection does not offer any advantage in applications that use connections for an extended period of time.

Locations for Specifying Pooling Parameters

For Java clients that use an IOM Bridge connection, specify pool parameters in one of the following locations:

- the source code. For details, see "Pooling with Directly Supplied Server Attributes " on page 19.
- a SAS Metadata Server. For details, see "Pooling with Server Attributes Read from a Metadata Server" on page 22.

Using Pooled Connections

When a request for a connection arrives, the request is handled as follows:

- If an existing pooled connection is available, then the Java Connection Factory returns that connection.
- If an existing pooled connection is not available, then the Java Connection Factory creates a new connection.

Users must notify the factory when they are finished with the connection so that it can be returned to the pool or destroyed.

The factory might have a limit on the number of connections it is allowed to create and manage at a time. If a factory has already allocated all of the connections that it can manage and a new connection request arrives, then the factory cannot serve the request immediately. You can specify how long to wait for another user to return a connection to the factory's pool.

Waiting for Connections to Become Available

At the time of a client's request for an object, it is possible that all of the available connections in a connection pool are already allocated to other clients. For example, the Java Connection Factory might not be able to make an additional connection to a server because it would exceed the **`sasMaxClients`** value that has been set for the server. In such cases, the client's request cannot be fulfilled until one of the other clients is finished with its object.

To indicate what action you want the Java Connection Factory to take when a request cannot be fulfilled immediately, you can use a **`long`** parameter with the **`getConnection`** method in the client program. The behavior of the Java Connection Factory depends on the value of the **`long`** parameter, as follows:

- If you specify a positive number, then the Connection Factory attempts to fulfill the request for that number of seconds before returning an exception.
- If you specify 0, then the Connection Factory attempts to fulfill the request for an unlimited amount of time.

- If you specify a negative number, then the Connection Factory attempts to fulfill the request once. If the request cannot be fulfilled immediately, then the Connection Factory returns an exception.

Pooling with Directly Supplied Server Attributes

Overview of Pooling with Directly Supplied Server Attributes

With just a few changes to the example program in "Connecting with Directly Supplied Server Attributes " on page 8, you can use the Java Connection Factory to manage a pool of connections to an IOM server rather than a single factory-managed connection.

When set up for connection pooling, the Java Connection Factory tries to fulfill each client's requests for connections by using an existing connection to an IOM server. This method is less time consuming than creating a new connection. Behind the scenes, the Java Connection Factory keeps a configurable number of connections alive at all times. For connection pooling to work properly, you must notify the Java Connection Factory when you are finished using a connection by calling **`close()`** on a factory-managed connection. For more details, see "Returning Connections to the Java Connection Factory " on page 24. When a client uses an object, it has exclusive access to the connection serving that object. When the client is finished using the object, the object is closed before the connection is returned to the pool. These actions help preserve the performance and security of the single connection case.

To create a pool, create the servers and then create the puddles.

You can maintain performance standards by spreading a pool of connections over more than one server and then setting an upper limit on the number of connections that each server can contribute to the pool. To specify multiple servers, provide a separate **`Server`** object for each server that is to participate in the pool. You can specify the following properties for each server (**`Server`** object) in addition to the other server properties described earlier.

MaxClients
: specifies the maximum number of connections that the pool will be allowed to make to the server at one time. Factors that you should consider when determining a value for this field include the number and type of processors on the machine, the amount of memory present, the type of clients that will be requesting objects, and the number of different pools the server participates in. This property is optional. The default value is 10.

RecycleActivationLimit
: specifies the number of times a connection to the server will be reused in a pool before it is disconnected (recycled). If the value is 0, then there will be no limit on the number of times a connection to the server can be reused. This property is optional. The default value is 0.

ServerRunForever
: must be either *true* or *false*. If the value is *false*, then unallocated live connections will be disconnected after a period of time (determined by the value of *ServerShutdownAfter*) unless they are allocated to a user before that period of time passes. Otherwise, unallocated live connections will remain alive indefinitely. This property is optional. The default value is *true*.

ServerShutdownAfter
: specifies the period of time, in minutes, that an unallocated live connection will wait to be allocated to a user before shutting down. This property is optional and it is ignored if the value of *ServerRunForever* is *true*. The value must not be less than 0, and it must not be greater than 1440 (the number of minutes in a day). The default value is 3. If the value is 0, then a connection returned to a pool by a user will be disconnected immediately unless another user is waiting for a connection from the pool.

A pool consists of one or more puddles (**`Puddle`** objects). A puddle is an association of one or more IOM servers with exactly one IOM login. In addition to the information about the servers that participate in a connection pool, you must specify information in order to create the puddles. You provide this information to the Java Connection Factory on the **`Puddle`** object. Here is a list of the properties that can be specified:

Credential
: specifies the login credential object that is associated with the puddle.

MinSize
: specifies the minimum number of connections that the Java Connection Factory can maintain for a puddle (after the initial start-up period). This number includes both the connections that are in use and the connections that are idle. This property is optional. The default value is 0.

MinAvail
: specifies the minimum number of idle connections that the Java Connection Factory can maintain for a puddle. This number includes only the connections that are idle. This property is optional. The default value is 0.

To specify multiple puddles, provide a separate **`Puddle`** object for each puddle that is to participate in the pool. You can then make a connection factory configuration with the list of puddles. For more details about supplying pooling and puddle information directly in the source code, see the Java API class documentation for the Java Connection Service at **`http://support.sas.com/rnd/javadoc/93`**.

Example

The following example demonstrates how to specify server properties to the Java Connection Factory and obtain four object references by using only two connections to IOM servers. In this example, the pool consists of a puddle with two servers. For information about how to use the object reference, see "Java Connection Factory Language Service Example" on page 15.

The last part of this example shows how to dispose of an object reference. For details about this procedure, see "Returning Connections to the Java Connection Factory " on page 24.

```
import java.util.HashSet;
import java.util.Set;
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SAS.IWorkspaceHelper;
import com.sas.services.connection.BridgeServer;
import com.sas.services.connection.Cluster;
import com.sas.services.connection.ConnectionFactoryAdminInterface;
import com.sas.services.connection.ConnectionFactoryConfiguration;
import com.sas.services.connection.ConnectionFactoryInterface;
import com.sas.services.connection.ConnectionFactoryManager;
import com.sas.services.connection.ConnectionInterface;
import com.sas.services.connection.Credential;
```

```
import com.sas.services.connection.LoadBalancingCluster;
import com.sas.services.connection.ManualConnectionFactoryConfiguration;
import com.sas.services.connection.PasswordCredential;
import com.sas.services.connection.Puddle;
import com.sas.services.connection.Server;
import org.omg.CORBA.Object;

// identify the IOM servers
String classID = Server.CLSID_SAS;
int port = 5310;
String domain = "unx";
Server server0 = new BridgeServer(classID,"serv1.unx.abc.com",port,domain);
Server server1 = new BridgeServer(classID,"serv2.unx.abc.com",port,domain);
Server[] servers = {server0,server1};
Cluster cluster = new LoadBalancingCluster(servers);

// create a login for these servers
Credential login = new PasswordCredential("adm1","adm1pass",domain);

// create a set of users allowed to use the connections to the servers
Credential user1 = new PasswordCredential("use1","use1pass");
Credential user2 = new PasswordCredential("use2","use2pass");
Set authorizedLogins = new HashSet(2);
authorizedLogins.add(user1);
authorizedLogins.add(user2);

// make a puddle with the servers
Puddle puddle = new Puddle(cluster,login);
puddle.setUserCredentials(authorizedLogins);

// make a connection factory configuration with the puddle
ConnectionFactoryConfiguration cxfConfig =
   new ManualConnectionFactoryConfiguration(puddle);

// get a connection factory that matches the configuration
ConnectionFactoryManager cxfManager = new ConnectionFactoryManager();
ConnectionFactoryInterface cxf =
   cxfManager.getFactory(cxfConfig); /* Use a private factory */

// cxfManager.getConnectionFactory(cxfConfig); /* Use a shared factory */

// get connections
ConnectionInterface cx1 = cxf.getConnection(user1);
Object obj1 = cx1.getObject();
IWorkspace iWorkspace1 = IWorkspaceHelper.narrow(obj1);
System.out.println(iWorkspace1.UniqueIdentifier());
ConnectionInterface cx2 = cxf.getConnection(user2);
Object obj2 = cx2.getObject();
IWorkspace iWorkspace2 = IWorkspaceHelper.narrow(obj2);
System.out.println(iWorkspace2.UniqueIdentifier());

< insert iWorkspace1 and iWorkspace2 usage code here >
cx1.close();
cx2.close();
ConnectionInterface cx3 = cxf.getConnection(user1);
Object obj3 = cx3.getObject();
```

```
IWorkspace iWorkspace3 = IWorkspaceHelper.narrow(obj3);
ConnectionInterface cx4 = cxf.getConnection(user1);
CORBA.Object obj4 = cx4.getObject();
IWorkspace iWorkspace4 = IWorkspaceHelper.narrow(obj4);

< insert iWorkspace3 and iWorkspace4 usage code here >

cx3.close();
cx4.close();

// tell the factory that it can destroy unused connections
admin.shutdown();
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Pooling with Server Attributes Read from a Metadata Server

When you connect to an IOM server using information from a metadata server, all of the information about the IOM server and how to connect to it is created and maintained by the metadata server administrator. The person developing the Java client application does not need to make a decision about whether to use connection pooling, because that decision is made by the metadata server administrator.

You can specify the pooling parameters with SAS Management Console. For details about planning for pooling and puddles on a SAS Metadata Server, see "Configuring Client-Side Pooling" in the *SAS Intelligence Platform: Application Server Administration Guide*. The code example in "Connecting with Server Attributes Read from a SAS Metadata Server " on page 10 can be used to connect to a pooled server without any changes. However, the credentials that you specify for the metadata server connection must be the credentials for the pooling administrator.

Note: If you are using the Information Service to obtain the server metadata, see "Pooling with Server Attributes Read from the Information Service" on page 22.

Pooling with Server Attributes Read from the Information Service

The process of creating a pooling server by using the Information Service is basically the same as the process that is described in "Connecting with Server Attributes Read from the Information Service " on page 12, but the following additional steps are required when you create a pooling server:

- create a user context for the pooling administrator user, who can access the metadata for the puddle logins. For more information about the pooling administrator, see "Configuring Client-Side Pooling" in the *SAS Intelligence Platform: Application Server Administration Guide*.
- specify the name or the user context for the pooling administrator in the constructor method for PlatformConnectionFactoryConnection.

The following example code shows how to create a pooling server using the Information Service. For information about how to use the object reference, see "Java Connection Factory Language Service Example" on page 15.

The last three statements in the example code show how to dispose of object references. For details about this procedure, see "Returning Connections to the Java Connection Factory " on page 24.

```
import com.sas.iom.SAS.IWorkspace;
import com.sas.iom.SAS.IWorkspaceHelper;
import com.sas.services.connection.ConnectionFactoryConfiguration;
import com.sas.services.connection.ConnectionInterface;
import com.sas.services.connection.platform.PlatformConnectionFactoryInterface;
import com.sas.services.connection.platform.PlatformConnectionFactoryConfiguration;
import com.sas.services.information.RepositoryInterface;
import com.sas.services.information.metadata.LogicalServerInterface;
import com.sas.services.user.UserContextInterface;
import com.sas.services.user.UserServiceInterface;

< set up the User Service and create a UserServiceInterface uService >

// Step 1. Create a user context for the pool administrator
UserContextInterface poolUser = uService.newUser("pooladm","pooladmpw",
   "pooladmdomain");

// Step 2. Create a user context for a connection factory user
UserContextInterface cxfUser = uService.newUser("user1","user1pw",
   "user1domain");

< set up the Information Service and define a repository repos >

// Step 3. Identify the repository
RepositoryInterface cxfRepos = cxfUser.getRepository("repos");

// Step 4. Identify the IOM service
LogicalServerInterface logicalServer =
   cxfRepos.fetch("A50IFJQG.AQ000002/LogicalServer");

// Step 5. Create a connection factory configuration
String poolUserName = poolUser.getName();
ConnectionFactoryConfiguration cxfConfig = new
   PlatformConnectionFactoryConfiguration(logicalServer, poolUserName);

// Step 6. Get a connection factory manager
PlatformConnectionFactoryManager cxfManager = new
   PlatformConnectionFactoryManager();

// Step 7. Get a connection factory using the configuration
PlatformConnectionFactoryInterface cxf =
   cxfManager.getPlatformFactory(cxfConfig);

// Step 8. Get a connection
ConnectionInterface cx = cxf.getConnection(cxfUser);

// Step 9. Narrow the connection
org.omg.CORBA.Object obj = cx.getObject();
IWorkspace iWorkspace = IWorkspaceHelper.narrow(obj);
```

```
< insert iWorkspace workspace usage code here >

// Step 10. Close the connection when finished
cx.close();
cxf.getAdminInterface().destroy();
```

Note: To make the previous example more readable, we have removed most of the code structuring elements. The example will not compile as it is shown.

Returning Connections to the Java Connection Factory

Closing a Connection to the Java Connection Factory

When you are finished using a connection that you have obtained from the Java Connection Factory, you must return the connection to the factory so that it can be either reused or canceled.

To return a connection to the Java Connection Factory, call the **`close()`** method on the connection that was returned when you called the **`getConnection`** method.

The objects that implement the **`IWorkspace`** interface also have a **`Close()`** method. You do not need to call this method when you are closing an object because the **`close()`** method on the connection object calls it for you.

If you do not explicitly close a connection, then it closes itself when it is no longer referenced and is garbage collected. However, you generally cannot determine when or if garbage collection will occur. Therefore, it is recommended that you explicitly close your connection if at all possible rather than depending on garbage collection.

Shutting Down the Java Connection Factory

When you are finished with the instance of the Java Connection Factory itself and you no longer need to request connections from it, you must shut it down so that any remaining connections can be canceled and other resources can be released.

To shut down the Java Connection Factory, call one of the following methods:

- The **`shutdown()`** method immediately cancels all idle connections in the pool. If connections are currently allocated to users, the connection factory waits and cancels these connections after the users return the connections to the factory. In addition, the Java Connection Factory will no longer honor new requests for connections. After **`shutdown()`** has been called, later calls to **`shutdown()`** have no effect.
- The **`destroy()`** method immediately cancels connections in the pool, including connections that have been allocated to users. Any attempt to use a connection from the factory will result in an exception. In addition, the Java Connection Factory will no longer honor new requests for connections. For user-managed connections, the **`destroy()`** method never destroys the connection. After **`destroy()`** has been called, later calls to **`shutdown()`** or **`destroy()`** have no effect.

It is often possible to cancel all connections and release all resources in an instance of the Java Connection Factory by calling **`shutdown()`** and being sure to call **`close()`** on all the connections. However, you can call **`destroy()`** instead of (or after) calling

`shutdown()` to ensure that an instance of the Java Connection Factory has been properly cleaned up.

Note: If you are using the PlatformConnectionFactory and the Session Service, you can shut down servers automatically by destroying a session. When you destroy a session, any repository connections associated with the session are destroyed. In addition, all connection factories that were configured with the repository connections are shut down as with the **`shutdown()`** method.

Chapter 3

Using Java CORBA Stubs for IOM Objects

Using Java CORBA Stubs for IOM Objects

This section describes some of the differences between Java client programming with CORBA and regular, non-distributed Java programming. This information should help you understand the more complex elements of Java client programming for the IOM server because the Java software for using the IOM server is based on CORBA standards.

CORBA is a set of standards defined by the Object Management Group (OMG) computer industry consortium that enables software objects to communicate with each other regardless of the language that is used to write the objects and the communications medium that is used to connect the objects. For more information, see the **`www.omg.org`** Web site.

In the Java client environment, there are two important parts of CORBA communication software: an object request broker (ORB) and stubs for IOM objects. The stubs are Java classes that have methods that correspond to the operations and attributes of a remote object. To invoke an operation on a remote object, you instantiate the appropriate stub and call the corresponding method. The stubs do not actually implement the functionality of remote objects. Rather, when the stubs receive a method call, they collect information about it (such as the method name and parameters), repackage that information into a request, and then forward the request through a Java ORB to the remote server that hosts the remote object.

A Java ORB is a library of Java classes that sends requests from a stub over a communications medium (typically a TCP/IP network) to an object that implements the method. The format of a request is specified in strict detail by the CORBA standard, but the way that an ORB sends a request over a network is not standardized. The CORBA standard does specify a protocol called Internet Inter-ORB Protocol (IIOP) that ORBs must use if they are to interoperate with other ORBs (perhaps created by other vendors). In the most common CORBA applications, a stub calls into an IIOP ORB, which communicates via IIOP with another IIOP ORB. The second ORB calls out to an object that implements the desired functionality. However, if interoperation with other ORBs is not a priority, then protocols other than IIOP can be used to send requests across a network.

SAS Integration Technologies features a Java ORB called the IOM Bridge for Java that communicates with the IOM server through a proprietary network protocol called IOM Bridge. Though the IOM Bridge for Java does not use IIOP, it does conform to the CORBA standard for the format of a request. SAS Integration Technologies also provides stubs for all of the IOM objects that are included in the IOM object hierarchy. The ORB and the stubs give you all you need to begin writing Java programs that can access the IOM Server.

Our ORB, the IOM Bridge for Java, is used internally by the Java Connection Factory and by the stubs, so you rarely need to know any details about the operation of the ORB or about its interface. However, the stubs collectively provide the primary interface for exploiting the functionality of the IOM server. Therefore, the Java programming information provided in this section deals with the use of IOM object stubs.

Null References

In Java programming, **`null`** can be assigned to any variable of a reference type (that is, a non-primitive type) to indicate that the variable does not refer to any object or array. CORBA also allows null object references, but it is important to note that not all Java reference types map to CORBA object references. Therefore, you might encounter situations where a null object reference that would be appropriate in a non-distributed Java program is not appropriate in a distributed Java program using CORBA. If **`null`** is used improperly in a method call on a Java CORBA stub, then the method will throw a **`java.lang.NullPointerException`**.

When calling methods on Java CORBA stubs like the IOM object stubs, **`null`** might be used in place of a reference to any Java object that implements **`org.omg.CORBA.Object`**. That means that **`null`** cannot be used in place of a reference to a Java object like an instance of **`java.lang.String`** or a Java array.

The **`GetApplication`** method on the Java CORBA IOM stub **`com.sas.iom.SAS.IWorkspace`** provides a good example. Here is the method signature for this method:

```
public
org.omg.CORBA.Object GetApplication
(
    java.lang.String application
)
throws
    com.sas.iom.SASIOMDefs.GenericError
```

When calling this method, the value of the parameter **`application`** cannot be **`null`** because its type, **`java.lang.String`**, does not implement

org.omg.CORBA.Object. However, the return value of the method can be **null** because the returned value does implement **org.omg.CORBA.Object**.

Exception Handling

Exception handling for Java clients for the IOM server is not significantly different from exception handling for any other Java program. Many methods in the stubs declare that they throw checked exceptions. When calling those methods, you must do so in a **try** block, and you must be sure to provide a **catch** block that handles each possible exception.

The stub documentation at **support.sas.com/rnd/gendoc/bi/api/iom/automj.html** provides information about why each exception is thrown and what to do when one is thrown.

Methods in the stubs can also throw unchecked exceptions when there is an error related to the distributed nature of your application. For example, an unchecked exception might be thrown when the communications subsystem fails or when the stubs are out of date relative to the IOM objects. All of these exceptions are subclasses of **org.omg.CORBA.SystemException**. A complete list of all subclasses is available in the CORBA specification. Because they are unchecked exceptions, the Java compiler does not require you to place your method calls inside a **try** block.

Output Parameters

CORBA includes the concept of output parameters, which are parameters that are uninitialized at the time of a call to a CORBA operation (CORBA operations map to Java methods), then initialized by the operation, and returned to the caller. Many IOM objects have operations that use output parameters.

Unfortunately, the concept of output parameters does not map well into Java. In Java method calls, parameters of primitive types are always passed by value and parameters of reference types are always passed by reference. In general, only the member variables of an object or elements of an array can be modified during a method call and returned to the caller. Furthermore, some objects are immutable, which means their members cannot be changed after the objects are constructed. Java CORBA programmers need a general way to use both primitive types and reference types for output parameters in method calls on Java CORBA stubs.

For this purpose, each data type that can be used for an output parameter in a method call on a Java CORBA stub is associated with a **Holder** class. A **Holder** class is a wrapper that has one public member variable of the targeted data type. When a **Holder** is used in a method call on a Java CORBA stub, the method implementation can set the member variable of the **Holder** to be the output value of the parameter, and the caller can fetch that value by getting the value of the member variable.

The value of the member variable in a **Holder** object before it is used in a method call with an output parameter is ignored, and, in the case of **Holder** classes for reference types, it can be **null**.

CORBA also includes the concept of update parameters, which are parameters that are initialized by the caller of a CORBA operation, possibly modified by the operation, and returned to the caller. In Java CORBA stubs, **Holder** classes are also used to handle update parameters.

For example, here is the definition of the class **org.omg.CORBA.IntHolder**, which is the **Holder** class for the Java primitive type **int**:

```
final public class IntHolder
{
   public int value;
   public IntHolder()
      {
      }
   public IntHolder(int initial)
      {
         value = initial;
      }
}
```

The following example shows how the **org.omg.CORBA.IntHolder** class could be used in a method call that requires an output **int** parameter:

```
org.omg.CORBA.IntHolder intHolder = new org.omg.CORBA.IntHolder();
myApplication.myMethod(intHolder);
int intValue = intHolder.value;
```

For a more practical use of **Holder** classes, see "Java Connection Factory Language Service Example" on page 15.

Generic Object References

When you obtain a reference to a stub for an IOM object, you usually call a method on another stub, and the stub takes care of the details necessary to connect the new stub with the new IOM object. However, sometimes a method is designed to produce a generic stub, which is a stub with no specialized methods.

Whenever a method on a stub has an output or return parameter of type **org.omg.CORBA.Object**, that parameter is considered a generic stub. Before you can do anything useful with a generic stub, you need to narrow it to a more specific stub.

Every stub is associated with a **Helper** class that contains a method called **narrow**. The **narrow** method converts a generic stub into a more useful one. If you attempt to narrow a generic stub to a specific stub that the underlying object cannot support, the **narrow** method returns **null**.

The following code fragment demonstrates the proper usage of narrowing:

```
org.omg.CORBA.Object generic =
   sasWorkspace.GetApplication(MY_APP);
IMyApp myApp = IMyAppHelper.narrow(generic);
myApp.myMethod();
```

IOM Objects That Support More than One Stub

Java CORBA stubs for IOM objects represent an interface that is implemented by the IOM object. Some IOM objects implement more than one interface, so you can use more than one stub to communicate with those objects. If you have a reference to a stub for one interface that an IOM object implements, then you can get a reference to a stub for any other interface that the IOM object implements using the **narrow()** method on the

Helper class for that stub. If you try to narrow an object reference to a stub for an interface that the IOM object does not implement, then the **narrow()** method returns **null**.

The following example uses the **Fileref** object, which implements the interfaces **com.sas.iom.SAS.IFileref** and **com.sas.iom.SAS.IFileInfo**:

```
com.sas.iom.SAS.IFileref iFileRef =
   sasFileService.UseFileref(MY_FILE);
com.sas.iom.SAS.IFileInfo iFileInsfo =
   com.sas.iom.SAS.IFileInfoHelper.narrow(iFileRef);
```

Events and Connection Points

Overview of Events and Connection Points

Some IOM objects support one or more event interfaces, which are interfaces that contain operations that are to be implemented by the client (in Java). The operations are called by the IOM object whenever some particular event occurs. For example, the SAS Language Component supports an event interface and calls operations on it whenever a SAS procedure or DATA step finishes execution, which enables you to check the progress of a submitted SAS program. To listen for events from an IOM object, you need to know how to use skeletons and connection points.

Extending Skeletons

A skeleton is the complement of a stub. While a stub is a Java class that repackages method calls into requests and forwards them to the IOM server, a skeleton is a Java class that accepts requests from the IOM server and repackages them into Java method calls. You provide the implementation of the method calls by extending the skeleton with implementations of all the methods in the event interface. When an event arrives, the IOM Bridge for Java provides a temporary thread of execution and calls the appropriate method through the skeleton.

The following example demonstrates how to extend the skeleton for the event interface supported by the SAS Language Component:

```
public class LanguageEventsListener extends
   com.sas.iom.SASEvents._ILanguageEventsImplBase
{
   // implement declared methods in com.sas.iom.SASEvents.ILanguageEvents
   public void ProcStart(java.lang.String procname) {
      /* your implementation */ }
   public void SubmitComplete(int sasrc) { /* your implementation */ }
   public void ProcComplete(java.lang.String procname) {
      /* your implementation */ }
   public void DatastepStart() { /* your implementation */ }
   public void DatastepComplete() { /* your implementation */ }
   public void StepError() { /* your implementation */ }
}
```

All of the methods return **void**, have only input parameters, and declare no exceptions. By definition, events do not produce any output and throw no checked exceptions, so when an IOM object sends an event, it is not obligated to wait for a response. If your implementation of a method in an event interface throws an unchecked exception, the

ORB catches it and ignores it. Furthermore, because no event requires output, you can provide trivial implementations for events that you are not interested in.

Finding a Connection Point

After you have written an event listener by using the preceding example as a guide, you then make the listener known to the IOM object by using a connection point. A connection point is, in effect, a child component of an IOM object that serves as a conduit for passing events from the IOM object to its listeners. IOM objects that support event interfaces implement an interface called **com.sas.iom.SASIOMDefs.ConnectionPointContainer**, which includes a method called **FindConnectionPoint()**. To call the **FindConnectionPoint()** method, you must narrow your object reference to **com.sas.iom.SASIOMDefs.ConnectionPointContainer**, as discussed in "Generic Object References" on page 30 and "IOM Objects That Support More than One Stub" on page 30..

The **FindConnectionPoint()** method provides you with a reference to the correct connection point. Because IOM objects can support more than one event interface, you must identify which connection point you want when you call **FindConnectionPoint()** by using the unique interface identifier of the event interface and the **com.sas.iom.SASIOMDefs.CP_ID** structure. The unique interface identifier for the event interface can be found by calling the **id()** method on the **Helper** class of the event interface.

The following example shows you how to get a unique interface identifier and use it to initialize the **com.sas.iom.SASIOMDefs.CP_ID** structure:

```
String cpidString = com.sas.iom.SASEvents.ILanguageEventsHelper.id();
int d1 = (int)java.lang.Long.parseLong(cpidString.substring(4,12),16);
short d2 = (short)java.lang.Integer.parseInt(cpidString.substring(13,17),16);
short d3 = (short)java.lang.Integer.parseInt(cpidString.substring(18,22),16);
byte[] d4 = new byte[8];

for (int i=0;i2;i++)
{
   d4[i] = (byte)java.lang.Short.parseShort(
      cpidString.substring(23+(i*2),25+(i*2)),16);
}

for (int i=0;i6;i++)
{
   d4[i+2] = (byte)java.lang.Short.parseShort(
      cpidString.substring(28+(i*2),30+(i*2)),16);
}

com.sas.iom.SASIOMDefs.CP_ID cpid=new com.sas.iom.SASIOMDefs.CP_ID(
   d1,d2,d3,d4);
```

After you have constructed the **com.sas.iom.SASIOMDefs.CP_ID** structure, you are ready to call **FindConnectionPoint()** and obtain a reference to the connection point component. Note that **FindConnectionPoint()** uses an output parameter to return a reference to the connection point, which means that you must use the **Holder** class **com.sas.iom.SASIOMDefs.ConnectionPointHolder**. Do not confuse that class with the **com.sas.iom.SASIOMDefs.ConnectionPointContainer** class.

The following example shows you how to find the connection point for the **com.sas.iom.SASEvents.ILanguageEvents** event interface:

```
com.sas.iom.SASIOMDefs.ConnectionPointContainer cpContainer =
   com.sas.iom.SASIOMDefs.ConnectionPointContainerHelper.narrow(sasLanguage);
com.sas.iom.SASIOMDefs.ConnectionPointHolder cpHolder =
   new com.sas.iom.SASIOMDefs.ConnectionPointHolder();
cpContainer.FindConnectionPoint(cpid,cpHolder);
com.sas.iom.SASIOMDefs.ConnectionPoint cp = cpHolder.value;
```

Using a Connection Point

After you have obtained a reference to a connection point, the final step is to make the connection point aware of your event listener. This step is done using the **`Advise()`** method. When you are no longer interested in receiving events, call the **`Unadvise()`** method.

The following example illustrates the use of a connection point:

```
org.omg.CORBA.IntHolder handleHolder = new org.omg.CORBA.IntHolder();
cp.Advise(sasLanguageEventsListener,handleHolder);
int handle = handleHolder.value;
/* event listener can now receive events */
cp.Unadvise(handle);
```

Datetime Values

Java, CORBA, and SAS all use different datetime formats. These formats are shown in the following table.

Table 3.1 *Datetime Formats*

Language	Starting Date	Increments
Java	January 1, 1970	milliseconds
CORBA	October 15, 1582	100s of nanoseconds
SAS	January 1, 1960	seconds

The IOM Bridge for Java and the Java CORBA stubs for IOM objects specify datetimes using the CORBA datetime format and a data type of **`long`**. To assist with conversions between Java and CORBA datetime formats, use the DateConverter class, as described in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**. Conversions between CORBA and SAS datetime formats are handled automatically by the IOM Bridge for Java.

Chapter 4

Using SAS Foundation Services

Overview of SAS Foundation Services

SAS Foundation Services is a set of infrastructure and extension services that support the development of integrated, scalable, and secure applications based on Java. SAS Foundation Services is based on the following design principles:

- implementation modularity
- location transparency
- robust and adaptive resource management
- run-time monitoring
- consistent deployment methodology
- client neutrality

The design model of SAS Foundation Services supports both local and remote resource deployment and promotes resource sharing among applications. Sharing can occur for a specific session, for a specific user, or globally, as appropriate. At the same time, the model controls access to protected resources based on privileged-user status and group membership.

SAS Foundation Services contains the following services:

For information about configuring and administering SAS Foundation Services, see the *SAS Foundation Services: Administrator's Guide*.

Connection Service

The Connection Service enables applications to do the following:

- connect to IOM servers that use the IOM Bridge Protocol
- use the Java Connection Factory to access existing connection objects and to create new connection objects for various server configurations
- use advanced connection management features such as connection pooling, failover, and load balancing, which are available through the Java Connection Factory

For detailed usage documentation and examples, see "Using the Java Connection Factory" on page 6 and **com.sas.services.connection.platform** in the Foundation Services class documentation at **http://support.sas.com/rnd/javadoc/93**.

Discovery Service

The Discovery Service enables applications to do the following:

- find implementations of SAS Foundation Services based on desired service capabilities and optional service attributes. Service capabilities are specified in terms of the Java interfaces that they implement. Discovery occurs without requiring the client to have any knowledge of the underlying lookup mechanisms that are being used.
- rediscover a previously discovered service by using its discovery service ID.

The Discovery Service can find service implementations that have been deployed locally for the application's exclusive use, as well as service implementations that have been deployed remotely for the use of multiple applications.

For detailed usage documentation and examples, see **com.sas.services.discovery** in the Foundation Services class documentation at **http://support.sas.com/rnd/javadoc/93**.

For information about deploying and configuring services either locally or remotely so that they can be found by Discovery Services, see "Understanding Service Deployments" in Chapter 1 of *SAS Foundation Services: Administrator's Guide* and "Understanding How Applications Locate Foundation Services " in Chapter 4 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide*.

Event Broker Service

The Event Broker Service enables applications to send and deliver events to the appropriate handling agents for processing. A handler can be either of the following:

- a statically defined process flow that runs in its own thread within the Event Broker Service to process the event. You can use the Foundation Services Manager plug-in to SAS Management Console to define the event and the process flow configuration.
- an application that has registered itself at run time with the Event Broker Service so that it can receive event notifications.

An Event Broker Service can also format a response to the processing of an event and send it as a reply to the event originator. It is the responsibility of the requester to specify the type of response that is desired: **`none`** (fire-and-forget), **`acknowledgement`** (acknowledge that the event was received), or **`result`** (send a formatted response).

An event is specified as a well-formed XML fragment that contains the name of the event, any associated properties, and a body.

For detailed usage documentation and examples, see **`com.sas.services.events.broker`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For details about editing the Event Broker Service configuration, see "Understanding the Event Broker Service" in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide*.

Information Service

The Information Service enables you to do the following:

- perform a combined search of all repositories that a user has a connection to. The classes in the Information Service package enable the creation of a single filter that can search multiple repositories.
- limit searches to a specific repository, so that efficient searching can be achieved.
- retrieve an item from a repository using a URL, using a convenience method.
- (in conjunction with the User Services and the Authentication Service) authenticate users, create User Contexts, locate servers that the user has access to, and create repository definitions to use in making server connections.

For detailed usage documentation and examples, see **`com.sas.services.information`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For information about configuring Information Services, see “Modifying the Information Service Configuration ” in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide*.

Logging Service

Note: The Logging Service is deprecated in SAS 9.3. You should use Log4j to perform logging tasks instead.

The Logging Service enables applications to do the following:

- send run-time messages to one or more output destinations, including consoles, files, and socket connections.
- configure and control the format of information sent to a particular destination. Configuration can be performed through static configuration files or by invoking run-time methods that control logging output.
- perform remote logging, which involves sending log messages generated in one Java virtual machine (JVM) to another JVM.
- perform logging either by user session or by JVM.

For detailed usage documentation and examples, see **`com.sas.services.logging`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For information about configuring a Logging Service, see “Modifying the Logging Service Configuration ” in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide*.

Publish Service

The Publish Service enables applications to do the following:

- create and populate collections of information that are called packages. Reports, tables, and documents are examples of the types of information that a package can contain.
- publish and retrieve packages using the following delivery transports:
  - archive transport, which is used to publish and retrieve binary archive files
  - channel transport, which is used to publish to a publication channel
  - requester transport, which is used to retrieve packages that are accessible by a SAS Workspace Server
  - WebDAV transport, which is used to publish to and retrieve from a WebDAV server
- generate a SASPackage Event, which contains information about a package that has been published.

For detailed usage documentation and examples, see **`com.sas.services.publish`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For information about the Publishing Framework, see the *SAS Publishing Framework: Developer's Guide*.

Security Service

The Security Service enables applications to do the following:

- authenticate the credentials of users. Authentication is the process of verifying that a user ID and its password are valid. The Security Service uses the Java Authentication and Authorization Service (JAAS) classes and interfaces to provide a pluggable authentication mechanism.
- propagate user identity contexts across distributed security domains.
- implement a single sign-on environment by saving credentials in the user context of an authenticated user.
- request a user's credentials from a user context that another application created. If credentials exist for the specified domain, then the application can use them to access resources without requiring additional authentication input from the user.

For detailed usage documentation and example code, see **`com.sas.services.security`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

Session Service

The Session Service enables applications to do the following:

- create a session context. A session context is a control structure that maintains state information within a bound session, facilitating resource management and context passing.
- bind objects to a session context.
- use the session context as a convenience container for passing multiple contexts.
- use the session context as a convenience container for passing other services, such as User Services and Logging Services.
- notify bound objects when they are removed from the session context or when the session context is destroyed, so that objects can perform any necessary cleanup.

For detailed usage documentation and examples, see **`com.sas.services.session`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For information about configuring a Session Service, see “Modifying the Session and User Service Configurations” in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide*.

Stored Process Service

The Stored Process Service enables applications to do the following:

- synchronously or asynchronously execute a stored process, which is a SAS language program that is stored on a SAS server. Execution can include accessing SAS data sources or external files and creating new data sets, files, or other data targets that are supported by SAS.
- receive values that have been assigned to input parameters and pass them to a stored process.
- return output from a stored process, either in a results package or in a streaming interface.

For detailed usage documentation and examples, see **`com.sas.services.storedprocess`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For information about SAS Stored Processes, see the *SAS Stored Processes: Developer's Guide*.

User Service

The User Service enables applications to do the following:

- create, locate, maintain, and aggregate information about users of SAS Foundation Services.
- store and retrieve user context objects for sharing between applications. The user context contains the user's active repository connections, identities, and profile.
- manage and access user profiles. A profile is a collection of name/value pairs that specify preferences and configuration or initialization data for a user for a particular application.
- access group profiles. A group profile specifies preferences and configuration or initialization data for a group of users for a particular application.

For detailed usage documentation and example code, see **`com.sas.services.user`** in the Foundation Services class documentation at **`http://support.sas.com/rnd/javadoc/93`**.

For information about configuring a User Service, see “Understanding Service Deployments” in Chapter 1 of *SAS Foundation Services: Administrator's Guide* and “Modifying the Session and User Service Configurations” in Chapter 5 of *SAS Foundation Services: Administrator's Guide* in the *SAS Foundation Services: Administrator's Guide*.

Chapter 5

Using JDBC Connections

Getting a JDBC Connection Object

Java Database Connectivity (JDBC) defines the standard way for Java programmers to access and manipulate data in a database. The IOM server supports IOM objects that provide all the functionality of a JDBC driver, but, instead of having to learn to program for those IOM objects, SAS Integration Technologies provides you with a JDBC implementation that uses IOM objects internally.

After you have established a connection to an IOM server and obtained a reference to a stub for the workspace component, you can get a **`java.sql.Connection`** object for the SAS JDBC Driver, as shown in the following example:

```
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;
import com.sas.iom.SAS.IDataService;
import com.sas.rio.MVAConnection;

//use Connection factory to get reference to workspace stub
//IWorkspace sasWorkspace = ...

IDataService rio = sasWorkspace.DataService();
Connection sqlConnection = new MVAConnection(rio,new Properties());
   .
   .
   .
(standard JDBC method calls)   .
   .
   .
```

The following example shows you how to release the connection:

```
sqlConnection.close();
```

Index

U