iWay

iWay Java Adapter for Mainframe Programming Guide

Version 5.1 Service Pack 1

DN3501654.0207

Preface

This documentation serves as a programming guide for iWay JAM .

How This Manual Is Organized

The following table lists each section of this manual with a brief description of its contents.

Chapter		**Contents**
1	Introduction to Generating Applications	Introduces generating applications and describes how iWay JAM uses DataViews and provides programmic access to services.
2	Generating a Java Application with eGen	Describes how to create a Java application from a COBOL copybook and a user-defined script file using the eGen Application Generator.
3	Basic Programming Techniques	Describes basic programming techniques.
4	Deploying Applications	Describes deploying an iWay JAM eGen EJB and an iWay JAM eGen servlet.
5	Understanding Programming Flows	Describes the interaction between WebLogic Server and mainframe programs.
6	Performing Your Own Data Translation	Describes performing your own data translation.
7	Diagnostics	Describes diagnostics and tracing.
A	DataView Programming Reference	Provides rules that enable you to identify what form a generated Java class takes from a given COBOL copybook processed by eGen Application Generator (eGen utility).
B	eGen Application Generator Reference	Contains reference pages for the iWay JAM eGen Application Generator (eGen utility), including the rules for writing the script file that controls the code generator.
C	Understanding How iWay JAM Uses XML	Describes how iWay JAM uses XML.
D	RMI Access to the iWay JAM Gateway	Describes RMI access to the iWay JAM Gateway and how to develop custom administrative capabilities.

Documentation Conventions

The following table lists and describes the conventions that apply throughout this manual.

Convention	Description
`THIS TYPEFACE` or `this typeface`	Denotes syntax that you must enter exactly as shown.
`this typeface`	Represents a placeholder (or variable) in syntax for a value that you or the system must supply.
`underscore`	Indicates a default setting.
this typeface	Represents a placeholder (or variable) in a text paragraph, a cross-reference, or an important term.
this typeface	Highlights a file name or command in a text paragraph that must be lowercase.
this typeface	Indicates a button, menu item, or dialog box option you can click or select.
Key + Key	Indicates keys that you must press simultaneously.
`{ }`	Indicates two or three choices; type one of them, not the braces.
`|`	Separates mutually exclusive choices in syntax. Type one of them, not the symbol.
`...`	Indicates that you can enter a parameter multiple times. Type only the parameter, not the ellipsis points (…).
`.` `.` `.`	Indicates that there are (or could be) intervening or additional commands.

Customer Support

Do you have questions about iWay JAM?

If you bought the product from a vendor other than iWay Software, contact your distributor.

If you bought the product directly from iWay Software, call Information Builders Customer Support Service (CSS) at (800) 736-6130 or (212) 736-6130. Customer Support Consultants are available Monday through Friday between 8:00 a.m. and 8:00 p.m. EST to address all your iWay JAM questions. iWay Software consultants can also give you general guidance regarding product capabilities and documentation. Please be ready to provide your six-digit site code number (*xxxx.xx*) when you call.

You can also access support services electronically, 24 hours a day, with InfoResponse Online. InfoResponse Online is accessible through our World Wide Web site, http://www.iwaysoftware.com. It connects you to the tracking system and known-problem database at the iWay Software support center. Registered users can open, update, and view the status of cases in the tracking system and read descriptions of reported software issues. New users can register immediately for this service. The technical support section of www.iwaysoftware.com also provides usage techniques, diagnostic tips, and answers to frequently asked questions.

To learn about the full range of available support services, ask your iWay Software representative about InfoResponse Online, or call (800) 969-INFO.

Help Us to Serve You Better

To help our consultants answer your questions effectively, please be prepared to provide specifications and sample files and to answer questions about errors and problems.

The following tables list the specifications our consultants require.

WebLogic Server Platform Operating System and Operating System Version	
WebLogic Server Version and Service Pack / special patch	
iWay JAM CRM Platform Operating System	
CICS or IMS Operating System and Operating System Version	
CICS or IMS Version Information	
Third party SNA stack product and version	

The following table lists components. Specify the version in the column provided.

Component	
iWay JAM Gateway Build / Fix Level	
iWay JAM CRM Build / Fix Level	
Pre-existing eGen Application Compiles? Specify Yes or No.	

In the following table, specify the JVM version and vendor in the columns provided.

Version	Vendor

The following table lists additional questions to help us serve you better.

Request/Question	Error/Problem Details or Information
Provide usage scenarios or summarize the application that produces the problem.	
Did this happen previously?	
Is this configuration working on any other system?	
Can you reproduce this problem consistently?	
Any **change in the application environment** including: • Migrating to a new WebLogic Server service pack or version. • Installing a new operating system on WebLogic Server side or CRM component side. • Migrating to a new CICS or IMS region, version, or operating system.	
Under what circumstance does the problem *not* occur?	
Describe the **steps** to reproduce the problem.	
Describe the **problem**.	

Request/Question	Error/Problem Details or Information
Specify the **error** message(s).	

The following table lists error/problem files that might be applicable.

Error/Problem Files	**Error/Problem File Detail or Information**
WebLogic Server Application logs or error messages	
CICS or IMS application logs or error messages	
WebLogic Server configuration file	
iWay JAM configuration file	
Third party stack SNA configuration	
VTAM Logical Unit definitions	
iWay JAM CRM startup script and script messages	
iWay JAM CRM JCL and JOB information (JESMSGLG, JESJCL, JEYSYSMSG)	
iWay JAM gateway trace diagnostics	
iWay JAM CRM trace diagnostics	
APPC trace files	

Collecting iWay JAM Diagnostics

If you are requested to collect additional iWay JAM diagnostic files, see the Diagnostics section in the *iWay Java Adapter for Mainframe Programming Guide*.

iWay JAM run-time traces are sent to the WebLogic log as “Debug” messages. Debug messages are written to each WebLogic Server log file but are not sent to the administration server. In addition, these messages are only sent to the server’s Stdout if the server configuration has both the *Log to Stdout* and *Debug to Stdout* options selected on the server Logging/General page.

For instructions on accessing Gateway tracing options, see the *iWay Java Adapter for Mainframe Configuration and Administration Guide*.

User Feedback

In an effort to produce effective documentation, the Documentation Services staff welcomes your opinions regarding this manual. Please use the Reader Comments form at the end of this manual to communicate suggestions for improving this publication or to alert us to corrections. You also can go to our Web site, http://www.iwaysoftware.com and use the Documentation Feedback form.

Thank you, in advance, for your comments.

iWay Software Training and Professional Services

Interested in training? Our Education Department offers a wide variety of training courses for iWay Software and other Information Builders products.

For information on course descriptions, locations, and dates, or to register for classes, visit our World Wide Web site, http://www.iwaysoftware.com or call (800) 969-INFO to speak to an Education Representative.

Interested in technical assistance for your implementation? Our Professional Services department provides expert design, systems architecture, implementation, and project management services for all your business integration projects. For information, visit our World Wide Web site, http://www.iwaysoftware.com.

Contents

CHAPTER 1

Introduction to Generating Applications

Topics:

This section provides an introduction to generating applications and describes how iWay JAM uses DataViews and provides access to services.

Overview

Integrating applications that run on the mainframe with applications that run within BEA WebLogic Server requires solving three significant problems:

- **Connectivity**. How can applications invoke each other when they are running on different hosts? iWay JAM provides software components that establish connections between your WebLogic and mainframe environments. These components are described in detail in the *iWay Java Adapter for Mainframe Configuration and Administration Guide*.
- **Data Transformation**. Java applications running in WebLogic Server use Java numeric representation and character encoding schemes. Applications running in the mainframe environment use different numeric and character encoding schemes. In order for applications running in these disparate environments to communicate, the data that is communicated must be transformed between these different representations.
- **Programmatic Access**. Java applications running in WebLogic Server require an Application Programming Interface (API) to access applications running in the mainframe environment. There also must be an API that allows Java applications to be accessed on behalf of mainframe application.

iWay JAM provides Java classes that transform data to and from the native binary data types of the mainframe. iWay JAM provides a software development tool that allows you to generate Java applications. These generated Java applications include data translation code (DataViews) that translates data between Java and mainframe data formats. These generated Java applications also contain the methods needed to invoke mainframe applications, or to be invoked by mainframe applications, in conjunction with iWay JAM.

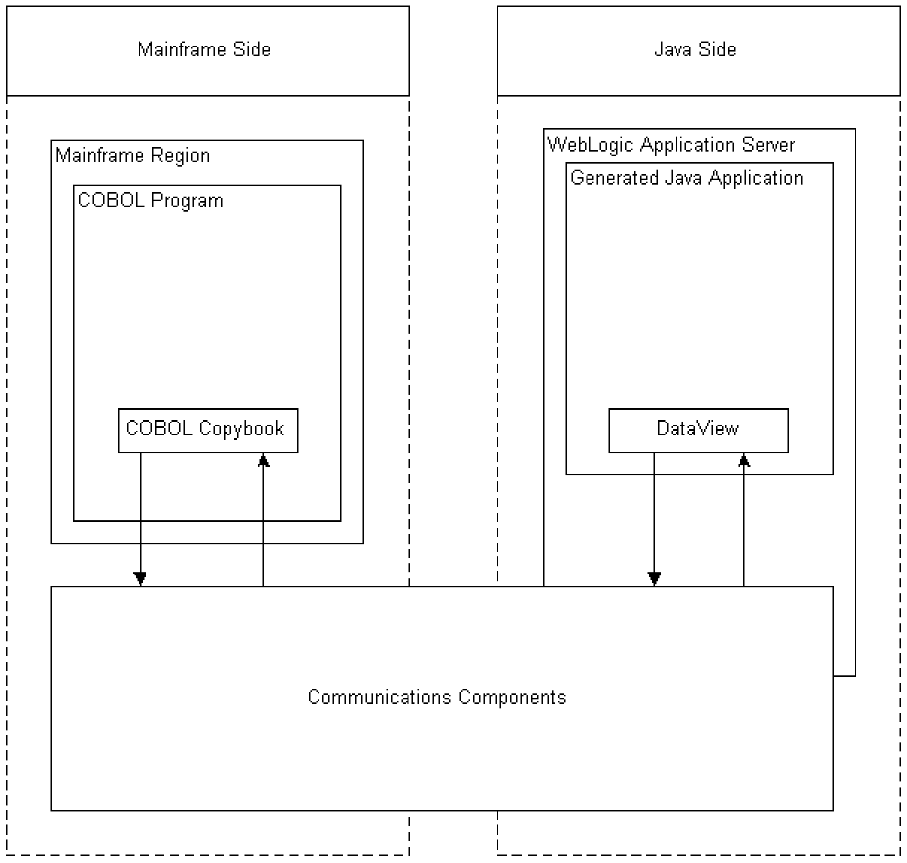
Understanding How iWay JAM Uses DataViews

In order to request services from the mainframe, iWay JAM must know the data formats required by these services. These data formats are usually available as COBOL copybooks.

Mainframe data records are represented in iWay JAM by Java DataViews. These DataViews are generated by the eGen Application Generator (hereafter referred to as the eGen utility) and provide all of the data translation necessary to communicate with mainframe applications. The eGen utility parses a COBOL copybook and generates Java DataView code that captures the data record described in the copybook. For more information on the eGen utility, see *Understanding eGen* in Chapter 2, *Generating a Java Application with eGen*.

The following diagram illustrates how iWay JAM uses DataViews. This illustration shows the COBOL copybook on the mainframe side, which contains the data formats for the mainframe services. When a request is made for a Java service, the data is passed through the communications components, which are described in more detail in the *iWay Java Adapter for Mainframe Introduction*. As part of this process, the iWay JAM Gateway initializes a DataView, performing the proper translation of the data. The data is utilized by the Java applications in the form of the DataView.

When the response is sent back, the iWay JAM Gateway translates the data back into the copybook format and sends it back to the mainframe.


Mainframe Side
Java Side
Mainframe Region
COBOL Program
COBOL Copybook
WebLogic Application Server
Generated Java Application
DataView
Communications Components

Understanding How iWay JAM Provides Programmatic Access to Services

Using iWay JAM, BEA WebLogic Server applications can make requests for mainframe services and receive responses to those requests. Applications in which these types of requests are made are referred to as WebLogic Server to Mainframe Applications. Also, mainframe applications can make requests from Java applications (EJBs) running in BEA WebLogic Server and receive responses to those requests. Applications in which these types of requests are made are referred to as Mainframe to WebLogic Server Applications.

iWay JAM provides an API that allows Java applications running under BEA WebLogic Server to invoke services running on the mainframe. All such requests for mainframe services are made by calling the callService() method of the EgenClient class. The Java applications generated by the eGen utility contain a method that calls the callService() method of the EgenClient class. These generated applications can access the callService() method by either being extensions of the EgenClient class or having an EgenClient class as a member. Instead of using the eGen utility to generate application code, you can also write your own applications that make requests of mainframe services by calling the callService() method, see *Chapter 6, Performing Your Own Data Translation*.

iWay JAM provides an API that allows clients running on the mainframe to invoke services provided by stateless session EJBs running under BEA WebLogic Server and receive responses to those requests. EJBs that can be invoked by iWay JAM on behalf of mainframe clients extend the EgenServerBean class. The iWay JAM Gateway calls the dispatch() method of the EgenServerBean class when a request is made from a mainframe client. The server EJBs generated by the eGen utility extend the EgenServerBean class. They also provide an implementation of the dispatch() method that includes the required data transformation, as well as making a call to the method that actually performs the business logic. You can write your own EJBs to service mainframe requests by extending the EgenServerBean class and implementing the dispatch() method.

iWay JAM also provides the ability for mainframe clients to queue messages on JMS queues and topics. No coding is necessary for this; it is simply a matter of configuration.

Application Model Overview

This guide provides four Java application models you can use as guides for creating your own applications. The following sections give you a brief overview of these models:

- Mainframe to WebLogic Server Application Models
- WebLogic Server to Mainframe Application Models

Mainframe to WebLogic Server Application Models

In a Mainframe to WebLogic Server application, a request originates from a mainframe and is serviced by an EJB invoked by a iWay JAM Gateway.

WebLogic Server to Mainframe Application Models

In a WebLogic Server to Mainframe application, a request originates on a WebLogic client or server, and is serviced by a mainframe program invoked by the iWay JAM Gateway in cooperation with the CRM.

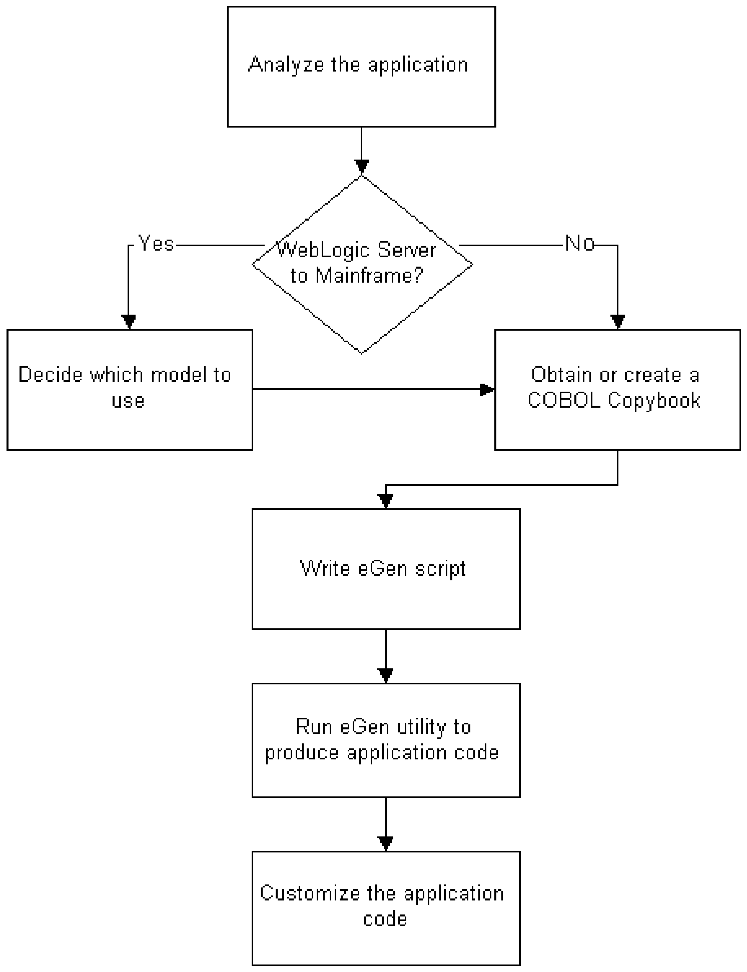
The following WebLogic Server to Mainframe application models are discussed in this guide:

- *Generating a Stand-Alone Client Application* in Chapter 3, *Basic Programming Techniques*.
- *Generating a Client Enterprise Java Bean-Based Application* in Chapter 3, *Basic Programming Techniques*.
- *Generating a Servlet Application* in Chapter 3, *Basic Programming Techniques*.

Roadmap for iWay JAM Programming

The steps outlined in the following diagram provide you with a high-level guideline to all of the tasks and processes that you must perform to generate applications using iWay JAM. You can think of these steps as a roadmap to guide you through the process and to point you to the resources available to help you.

The following diagram shows a roadmap for JAM programming.


Analyze the application
WebLogic Server to Mainframe?
Yes
No
Decide which model to use
Obtain or create a COBOL Copybook
Write eGen script
Run eGen utility to produce application code
Customize the application code

1. Analyze the application and determine if it is Mainframe to WebLogic Server or WebLogic Server to Mainframe. If the application is WebLogic Server to Mainframe, decide which model you are going to use, see *WebLogic Server to Mainframe Application Models* on page 1-5 for more information.
2. Obtain or create a COBOL copybook (see *Working With COBOL Copybooks* in Chapter 2, *Generating a Java Application with eGen* for more information.
3. Write the eGen script. The eGen script has two parts. The first part defines the DataView. The second part defines the application code, see *Writing an eGen Script* in Chapter 2, *Generating a Java Application with eGen* for more information.

4. Use the COBOL copybook and the eGen script as input for the eGen utility. This produces the DataView and the application code. For more information, *Processing eGen Scripts with the eGen Utility* in Chapter 2, *Generating a Java Application with eGen*.
5. Customize the application code. This can be done by extending the code to perform the tasks required for your application. For more information, see Chapter 3, *Basic Programming Techniques*.

CHAPTER 2

Generating a Java Application with eGen

Topics:

- Understanding eGen
- Working With COBOL Copybooks
- Writing an eGen Script
- Writing the DataView Section of an eGen Script
- Processing eGen Scripts with the eGen Utility

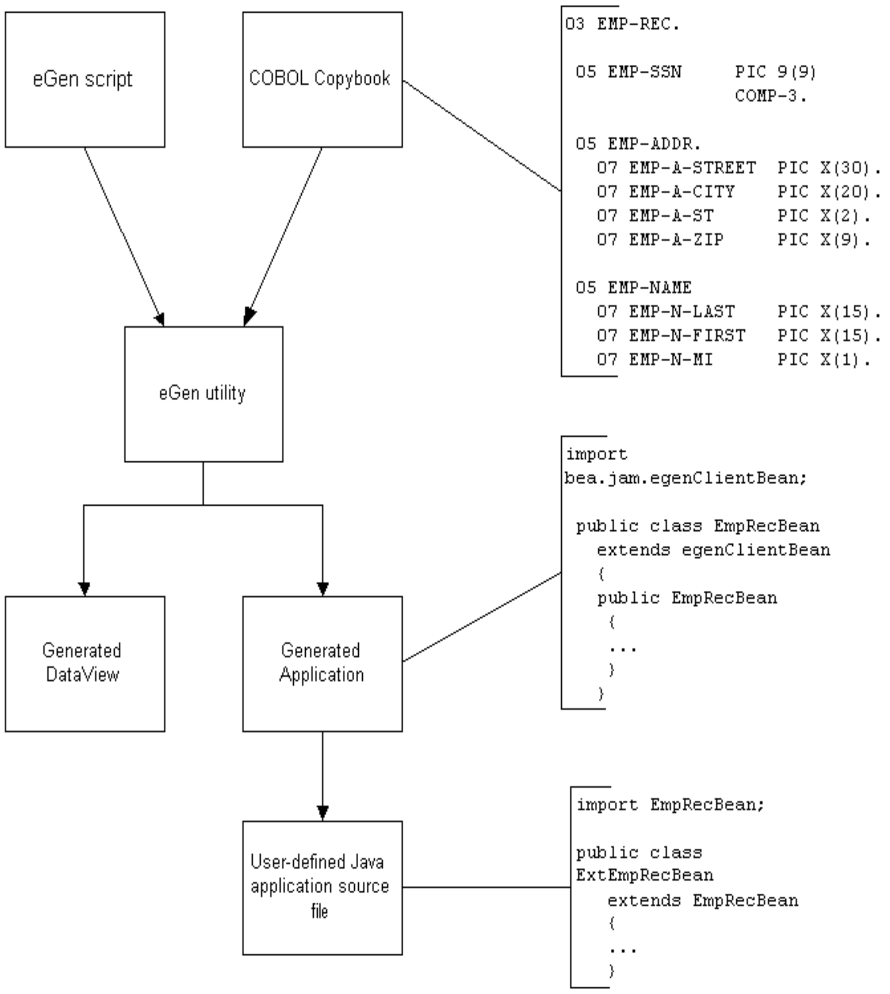
This section describes how to create a Java application from a COBOL copybook and a user-defined script file using the eGen Application Generator.

Understanding eGen

The eGen Application Generator, also known as the eGen utility, is installed with iWay JAM. It generates Java applications from a COBOL copybook and a user-defined script file.

The eGen utility generates a Java application by processing a script you create, called an eGen script. A Java DataView is defined by the first section of the script. This DataView is used by the application code to provide data access and conversions, as well as to perform other miscellaneous functions. The actual application code is defined by the second section of the script.

The following diagram shows how the eGen utility works. This diagram shows the eGen script and COBOL copybook file being used as input to the eGen utility, and the output that is generated is the DataView and the Java application. The generated Java application may be used in a variety of ways. In some cases, it may be used as is. However, in most cases, you must extend the generated application in some way, or it may become a member of the actual user-defined application.



```
03 EMP-REC.

 05 EMP-SSN      PIC 9(9)
                 COMP-3.

 05 EMP-ADDR.
   07 EMP-A-STREET  PIC X(30).
   07 EMP-A-CITY    PIC X(20).
   07 EMP-A-ST      PIC X(2).
   07 EMP-A-ZIP     PIC X(9).

 05 EMP-NAME
   07 EMP-N-LAST    PIC X(15).
   07 EMP-N-FIRST   PIC X(15).
   07 EMP-N-MI      PIC X(1).
```

```
import
bea.jam.egenClientBean;

 public class EmpRecBean
   extends egenClientBean
   {
   public EmpRecBean
    {
    ...
    }
   }
```

```
import EmpRecBean;

public class
ExtEmpRecBean
   extends EmpRecBean
   {
   ...
   }
```

Working With COBOL Copybooks

A COBOL CICS or IMS mainframe application typically uses a copybook source file to define its data layout. This file is specified in a COPY directive within the LINKAGE SECTION of the source program for a CICS application, or in the WORKING-STORAGE SECTION of an IMS program. If the CICS or IMS application does not use a copybook file, you must create one from the data definition contained in the program source.

The content of each copybook is parsed by the eGen utility, producing DataView sub-classes that provide facilities to:

- Convert COBOL data types to and from Java data types.

  This includes conversions for mainframe data formats and code pages.
- Convert COBOL data structures to and from Java data structures.
- Convert the provided data structures into other arbitrary formats.

The eGen utility must have a COBOL Copybook to use as input. There are two methods you can use to obtain this Copybook:

- Creating a New COBOL Copybook
- Using an Existing COBOL Copybook

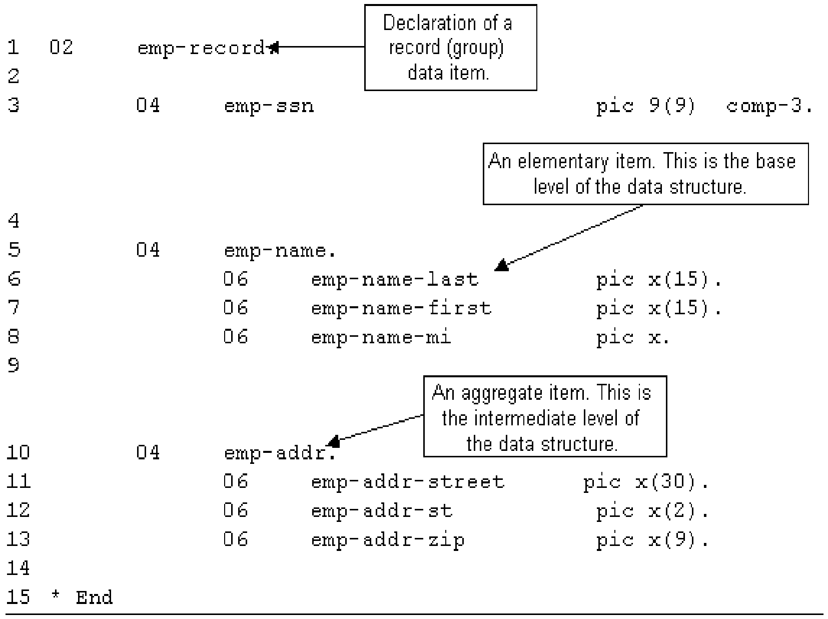
Creating a New COBOL Copybook

If you are producing a new application on the mainframe or modifying one, then one or more new copybooks may be required. You should keep in mind the COBOL features and data types supported by iWay JAM as you create these copybooks. See Appendix B, *eGen Application Generator Reference* for more information.

Using an Existing COBOL Copybook

When a mainframe application has an existing DPL or APPC interface, the data for that interface is usually described in a COBOL copybook. Before using an existing COBOL Copybook, verify that the interface does not use any COBOL features or data types that iWay JAM does not support. See *Limitations of the eGen Utility* on page 2-5.

An example COBOL copybook source file is shown in the following image.



```
02     emp-record

       04     emp-ssn                             pic 9(9)  comp-3.


       04     emp-name.
              06     emp-name-last          pic x(15).
              06     emp-name-first         pic x(15).
              06     emp-name-mi            pic x.


       04     emp-addr.
              06     emp-addr-street      pic x(30).
              06     emp-addr-st            pic x(2).
              06     emp-addr-zip           pic x(9).

* End
```

Declaration of a record (group) data item.
An elementary item. This is the base level of the data structure.
An aggregate item. This is the intermediate level of the data structure.

Limitations of the eGen Utility

The eGen utility can translate most COBOL copybook data types and data clauses into their Java equivalents; however, it is unable to translate some obsolete constructs and floating point data types. For information on COBOL data types that can be translated by the eGen utility, see Appendix A, *DataView Programming Reference*. If the eGen utility is unable to fully support constructs or data types, it:

- Treats them as alphanumeric data types (if reasonable).
- Ignores them (if their support is unimportant to iWay JAM operation).
- Reports them as errors.

If the eGen utility reports constructs or data types as errors, you must modify them, so they can be translated.

Writing an eGen Script

After you have obtained a COBOL Copybook for the mainframe applications, you are ready to write an eGen script. This eGen script and the COBOL copybook that describes your data structure is processed by the eGen utility to generate a DataView and application code which serves as the basis for your custom Java application.

An eGen script has the following two sections.

- DataView

  The DataView section of the script generates Java DataView code from a COBOL copybook. The class file compiled from the generated code extends the Java DataView class. Generating DataViews is discussed in detail in the remainder of this section.

  If the purpose of your eGen script is to generate a DataView for use with the iWay JAM to JMS EJB, or to launch a WebLogic Integration event, you are required to create only the DataView section of the script.

- Java application

  The Java application section of the script generates the Java application code. This is discussed in detail in Chapter 3, *Basic Programming Techniques*.

Writing the DataView Section of an eGen Script

The eGen utility parses a COBOL copybook and generates Java DataView code that encapsulates the data record declared in the copybook. It does this by parsing an eGen script file containing a DataView definition similar to the following example (keywords are in bold). The section containing the DataView definition is the first section of the eGen script. Application code is generated by the second section.

```
generate view examples.CICS.outbound.gateway.EmployeeRecord from emprec.cpy
```

where:

`generate view`

Tells the eGen utility to generate a Java DataView code file.

`examples.CICS.outbound.gateway.EmployeeRecord`

Tells the eGen utility to call the DataView file EmployeeRecord.java. The package is called examples.CICS.outbound.gateway. The EmployeeRecord class defined in EmployeeRecord.java is a subclass of the DataView class. The phrase from emprec.cpy tells the eGen utility to form the EmployeeRecord DataView file from the COBOL copybook, emprec.cpy.

Additional generate view statements may be added to an eGen script in order to produce all the DataViews required by your application. Also, additional options may be specified in the eGen script to change details of the DataView generation. For example, the following script generates a DataView class that uses code page, cp500, for conversions to and from mainframe format. If the code page clause is not specified, the default codepage of cp037 is used.

The following is an example of a DataView section with a code page specified.

```
generate view examples.CICS.outbound.gateway.EmployeeRecord from
emprec.cpy codepage cp500
```

The following script generates additional output intended to support use of the DataView class with XML data:

```
generate view sample.EmployeeRecord from emprec.cpy support xml
```

Additional files generated for XML support are listed in the following table which lists files for DataView XML support and their purposes

.

File Name	File Purpose
classname.dtd	XML DTD for XML messages accepted and produced by this DataView.
classname.xsd	XML schema for XML messages accepted and produced by this DataView.

Processing eGen Scripts with the eGen Utility

After you have written your eGen script, you must process it to generate the DataView and application code. This Java code must then be compiled and deployed. The same eGen script usually contains both the definitions of the DataView and application code, and both are produced with a single processing of the script. However, in this programming guide, the script is explained in two steps, so the code generated can be analyzed in greater detail.

Creating an Environment for Generating and Compiling the Java Code

When you process the eGen scripts and compile the generated Java code, you must have access to the Java classes and applications used in the code generation and compilation processes. Adding the correct elements to your CLASSPATH and PATH environment variables provides this access.

For the eGen utility:

- Add <JAM_INSTALL_DIR>\lib\iwjam.jar to your class path.
- Add <JAM_INSTALL_DIR>\bin to your path.

For compilation:

- Add <JAM_INSTALL_DIR>\lib\iwjam.jar to your class path.
- Add <WLS_HOME>\lib\weblogic.jar to your class path.
- Add the path of your DataView class files to your class path.

  You must access these classes when you compile your Java application code.

Note: UNIX users must use "/" instead of "\" when adding directory paths as specified above. Running config\verify\setVerifyEnv.cmd (on Windows systems) or config/verify/setVerifyEnv.sh (on UNIX systems) performs the previous actions that are required for the eGen utility.

Generating the Java DataView Code

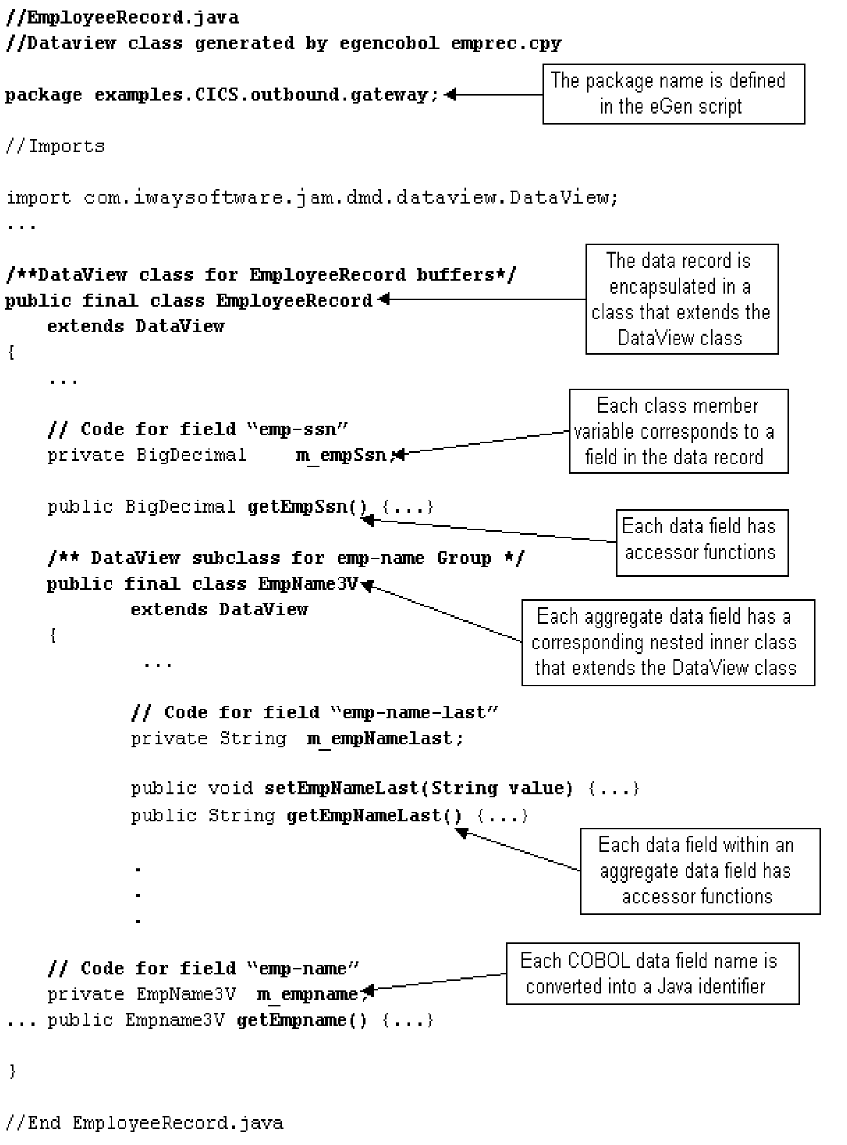
For the eGen script named emprec.egen shown in the following example, the shell command parses the copybook file named emprec.cpy and generates the EmployeeRecord.java source file in the current directory.

```
egencobol emprec.egen
```

If no error or warning messages are issued, the copybook is compatible with iWay JAM and the source files are created. No application source files are generated by processing the emprec.egen script. This is because there are no application generating commands in this script.

For suggestions on resolving any problems encountered, see Appendix B, *eGen Application Generator Reference*.

The following example illustrates the resulting generated Java source file, EmployeeRecord.java with some comments and implementation details removed for clarity.



```
//EmployeeRecord.java
//Dataview class generated by egencobol emprec.cpy

package examples.CICS.outbound.gateway;

//Imports

import com.iwaysoftware.jam.dmd.dataview.DataView;
...

/**DataView class for EmployeeRecord buffers*/
public final class EmployeeRecord
    extends DataView
{
    ...

    // Code for field "emp-ssn"
    private BigDecimal     m_empSsn;

    public BigDecimal getEmpSsn() {...}

    /** DataView subclass for emp-name Group */
    public final class EmpName3V
            extends DataView
    {
             ...

            // Code for field "emp-name-last"
            private String  m_empNamelast;

            public void setEmpNameLast(String value) {...}
            public String getEmpNameLast() {...}

            .
            .
            .

    // Code for field "emp-name"
    private EmpName3V  m_empname;
... public Empname3V getEmpname() {...}

}

//End EmployeeRecord.java
```

The package name is defined in the eGen script
The data record is encapsulated in a class that extends the DataView class
Each class member variable corresponds to a field in the data record
Each data field has accessor functions
Each aggregate data field has a corresponding nested inner class that extends the DataView class
Each data field within an aggregate data field has accessor functions
Each COBOL data field name is converted into a Java identifier

Special Considerations for Compiling the Java Code

You must compile the Java code generated by the eGen utility. However, there are some special circumstances to consider. Because the application code is dependent on the DataView code, you must compile the DataView code and ensure that the resulting DataView class files are in your environment class path before compiling your application code. You must ensure that all of the DataView class files can be referenced by the application code compilation.

For example, the compilation of EmployeeRecord.java results in the following class files:

```
EmployeeRecord.class

EmployeeRecord$EmpRecord1V.class

EmployeeRecord$EmpRecord1V$EmpName3V.class

EmployeeRecord$EmpRecord1V$EmpAddr7V.class
```

All of these class files are used when compiling your application code.

CHAPTER 3

Basic Programming Techniques

Topics:

- Choosing an eGen Java Application Model
- General Form of an eGen Script
- Mainframe to WebLogic Server Application Models
- WebLogic Server to Mainframe Application Models
- Supplying Security Credentials
- iWay JAM to JMS

This section describes basic programming techniques.

Choosing an eGen Java Application Model

There are four different types or models of Java applications that can be generated by the eGen utility. These models are classified as either Mainframe to WebLogic Server or WebLogic Server to Mainframe.

Mainframe to WebLogic Server. The request originates on the mainframe and is serviced by WebLogic Server.

- Server EJB

  The server EJB is a Stateless Session EJB that provides a service to the mainframe.

WebLogic Server to Mainframe. The request originates on the WebLogic client or server and is serviced by the mainframe:

- Client Class

  The client class is a stand-alone Java class that invokes mainframe services.

  This class may be built into your own EJB or utilized in some other way within your code.

- Client EJB

  The client EJB is a Stateless Session EJB that invokes mainframe services. It may be called by a servlet or other client programs.

  This is the normal model for building a production application with access to mainframe services. A servlet that invokes the EJB methods may be added for testing or demonstration purposes.

- Servlet Only

  The servlet-only application is a servlet that presents a simple form and invokes mainframe services directly.

  This is the simplest model, but it may not be suitable for production applications.

Choose one of these four model types to use as the basis for your Java application. After you chose a model type, refer to the topic from the following list for instructions for writing the script and implementing the model you chose.

- *Generating a Server Enterprise Java Bean-Based Application* on page 3-6.
- *Generating a Stand-Alone Client Application* on page 3-13.
- *Generating a Client Enterprise Java Bean-Based Application* on page 3-18.
- *Generating a Servlet Application* on page 3-26.

For all of the applications you generate, you must provide a script file containing definitions for the application, including the COBOL copybook file name and the DataView class names.

Generating the Java Application Code

The Java application code can be generated at the same time that you generate the Java DataView code. To generate Java application code, the eGen script that you process must contain instructions for generating the Java application along with the instructions for generating the DataView code.

Referring to the sample files in samples\verify\gateway\outbound, the following command generates Chardata.java and BaseClient.java. The DataView file is Chardata.java, and the application file is BaseClient.java.

```
> egencobol baseClient.egen
```

General Form of an eGen Script

As previously stated, most eGen scripts consist of the following major sections:

- The DataView section described in *Writing an eGen Script* in Chapter 2, *Generating a Java Application with eGen*.
- The Application section, which defines the Java application code that the eGen utility is to generate (described in *Writing the Application Section of an eGen Script* on page 3-3).

Writing the Application Section of an eGen Script

The application section of an eGen script contains the information about the Java class files that the eGen utility is to generate for a particular application. The application section is divided into two distinct subsections, which are actually lists. The two lists are:

- **List of Services.** Describes the remote services that are configured for JAM and are called by the classes that the eGen script defines. This list is not present in the script if the classes to be generated by the eGen utility are all server EJBs.
- **List of Application Components**. Components for which the eGen utility is to generate the class files. This list contains one or more definitions of stand alone clients, client EJBs, servlets, or server EJBs.

Reference: List of Services

Scripts that are used to define the application components that the eGen utility is to generate usually contain a list of one or more service definitions. If the application components are all server or Mainframe to WebLogic Server EJBs, this list of services is not present. This is because this list of service definitions describes remote services configured in JAM; server EJBs do not call remote services since the requests are flowing outward from the mainframe.

The general form of a service definition is as follows (keywords are in bold):

```
service servicename accepts inputViewname returns outputViewname
```

The following table lists and describes the service definition parameters.

Parameter	**Definition**
servicename	Must match the name of a remote service that is defined in the iWay JAM configuration (see the *iWay Java Adapter for Mainframe Configuration and Administration Guide*).
inputViewname	Name of a DataView that is the input or request data for the service.
outputViewname	Name of a DataView that is the output or response from the service.

The inputViewname and outputViewname are not required to be the same. However, because of the way many applications are written, they often are the same.

The following is an example of a service definition:

```
service TOUPPER accepts Chardata returns Chardata
```

In this example, the service TOUPPER is a configured remote service. For the purposes of the Java application making the request for a mainframe service through iWay JAM, this service accepts as input a Chardata DataView. The actual mainframe server application accepts as input the COBOL copybook which corresponds to a Chardata DataView. For the purposes of the Java application, the output or response from the mainframe service is a Chardata DataView.

Reference: List of Application Components

In order for the eGen utility to generate code for Java applications, the eGen script must contain a list of one or more definitions of the application components that are to be generated. This list of definitions of application components can contain definitions of stand-alone clients, client or server EJBs, and servlets. This list of definitions also contains the definition of the HTML pages that are used by servlets defined in the list.

Note: The definition of an HTML page appearing in this list by itself does not cause code to be generated.

The general form of an application component definition is as follows:

```
model identifier [model-dependent-parameters]
{ details }
```

The following table lists and describes the application component definition parameters.

Parameter	**Definition**
model	Indicates to the eGen utility the type of application component to generate. The possible values of this identifier are: • client class • client ejb • server ejb • servlet • page
identifier	Usually, the class name (or class name stem for EJBs) for the application component that is to be generated. The identifier includes the package name. For an HTML page, the identifier is the page name.
model-dependent parameters	Further describe the application component to the eGen utility and can vary depending on the model. For a stand-alone client, no model-dependent-parameters are given. For an EJB (client or server), the home interface identifier for the bean must be given. For a servlet, the initial HTML page to display is given. For an HTML page, the title of the page is given.
details	Details about the code for the application component. For a stand-alone client, as well as an EJB, these details include the definitions of class methods that call services defined in the script. For a servlet, usually no details are given. For an HTML page, the details include the DataView to display and any buttons to display on the page.

The following is an example of an application component definition, in this case, the definition of a client or EJB.

```
client ejb sample.SampleClient my.sampleBean
 {
        method newEmployee
                is service sampleCreate
 }
```

where:

`SampleClient`

Is the class name for this EJB, that is, the eGen utility generates files named SampleClient.java, SampleClientBean.java, and SampleClientHome.java.

`sample`

Is the package name.

`my.sampleBean`

Is the home interface identifier for the bean.

`newEmployee`

Is the method that calls the sampleCreate service. The sampleCreate service is defined elsewhere in the file.

Specific details about the application component definitions for each application model, as well as the files that the eGen utility generates for each model, are discussed in the following topics.

Mainframe to WebLogic Server Application Models

In a Mainframe to WebLogic Server application, a request originates on a mainframe and is serviced by an EJB invoked by an iWay JAM Gateway.

Generating a Server Enterprise Java Bean-Based Application

This type of application produces Java classes that comprise an EJB application acting as a remote server from the viewpoint of the mainframe. The classes process service requests originating from the mainframe (remote) system and transfer data records to and from the mainframe. From the viewpoint of the Java classes, they receive EJB method requests. From the viewpoint of the mainframe application, it invokes remote CICS or IMS programs.

Reference: Components of an eGen Server EJB Script

The general form of a definition of a server (Mainframe to WebLogic Server) EJB that appears in an eGen script is as follows (keywords are in bold):

```
server ejb classname ejbregistration transaction
   transaction-attribute
{servermethod}
```

The following table lists and describes the service EJB definition keywords and parameters.

Keyword/Parameter	Definition
server ejb	Indicates to the eGen utility the type of application component that is to be generated.

Keyword/Parameter	**Definition**
classname	Indicates the class name stem for the EJB. For example, if the class name is SampleServer, then the following files are generated by the eGen utility: • SampleServer.java • SampleServerBean.java • SampleServerHome.java **Note**: The package name should be included in the class name.
ejbregistration	The name used to register the home interface for the EJB.
transaction transaction- attribute	Optional. Used to manage the level of transaction demarcation. The possible values of the transaction-attribute are: • NotSupported • Required • Supports • RequiresNew • Mandatory • Never **Note**: If the transaction keyword is not present in the definition, the default value of the transaction-attribute is Supports. For a detailed explanation of how the WebLogic Server EJB container responds to the transaction-attribute setting, see the section on Transaction Attributes in the EJB 2.0 Specification.
servermethod	Method that appears in the EJB implementation (must be in braces). The general form of a servermethod definition is as follows (keywords are in bold): `method methodname (inputDataView) returns outputDataView`

The following table describes the parameters of a servermethod definition.

Parameter	Definition
methodname	Name of the method.
inputDataView	Name of the DataView that is the type of the input parameter for the method (must be in parenthesis).
outputDataView	Name of the DataView that is the type returned from the method.

The following is an example of a server (Mainframe to WebLogic Server) EJB definition that appears in an eGen script. This is the definition for a server EJB class. The generated EJB class files are defined in *Generated Files* on page 3-8.

```
server ejb sample.SampleServer my.sampleServer
{
method newEmployee (EmployeeRecord)
returns EmployeeRecord
}
```

where:

`my.sampleServer`

Is the home interface identifier for this bean in the WebLogic deployment description.

`server class method newEmployee`

Takes its input from the DataView EmployeeRecord and writes the output to an EmployeeRecord output DataView.

The transaction keyword is not present in this example, so it defaults to Supports.

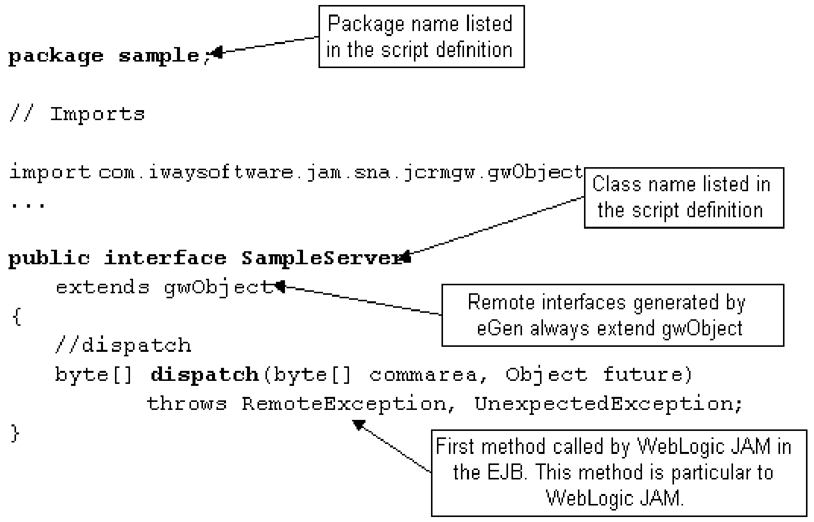
Reference: Generated Files

The following table lists the files generated from the example server (Mainframe to WebLogic Server) EJB described in *Components of an eGen Server EJB Script* on page 3-6. These files are described in the topics that follow the table.

File	Content
SampleServer.java	Source for the EJB remote interface.
SampleServerBean.java	Source for the EJB implementation.
SampleServerHome.java	Source for the EJB home interface.
SampleServer-jar.xml	Deployment descriptor.
wl-SampleServer-jar.xml	WebLogic deployment information.

SampleServer.java Source File

The following image shows the partial contents of the generated remote interface SampleServer.java source file.

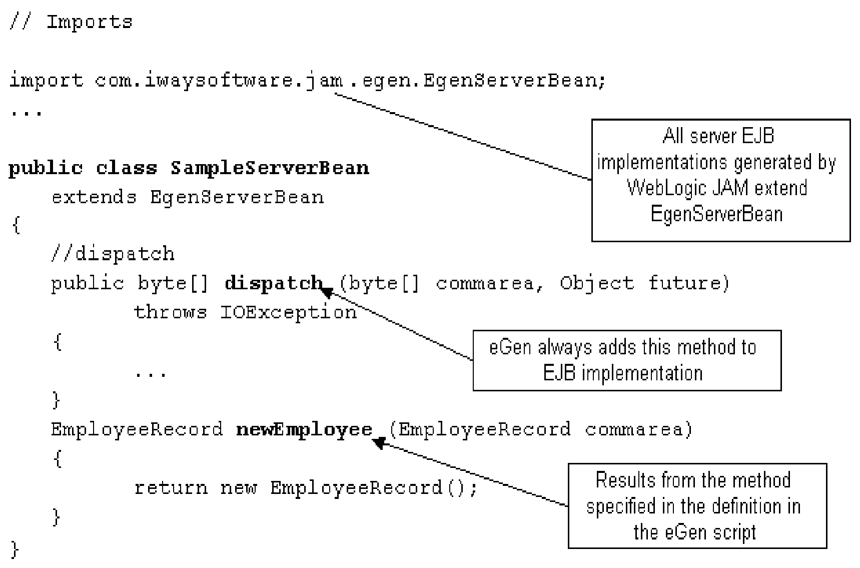


```
package sample;

// Imports

import com.iwaysoftware.jam.sna.jcrmgw.gwObject
...

public interface SampleServer
    extends gwObject
{
    //dispatch
    byte[] dispatch(byte[] commarea, Object future)
           throws RemoteException, UnexpectedException;
}
```

Package name listed in the script definition
Class name listed in the script definition
Remote interfaces generated by eGen always extend gwObject
First method called by WebLogic JAM in the EJB. This method is particular to WebLogic JAM.

SampleServerBean.java Source File

The following image shows the partial contents of the generated EJB implementation SampleServerBean.java source file.



```
// Imports

import com.iwaysoftware.jam .egen.EgenServerBean;
...

public class SampleServerBean
    extends EgenServerBean
{
    //dispatch
    public byte[] dispatch (byte[] commarea, Object future)
            throws IOException
    {
            ...
    }
    EmployeeRecord newEmployee (EmployeeRecord commarea)
    {
            return new EmployeeRecord();
    }
}
```

All server EJB implementations generated by WebLogic JAM extend EgenServerBean
eGen always adds this method to EJB implementation
Results from the method specified in the definition in the eGen script

SampleServerHome.java Source File

The eGen utility generates a standard home interface class for the server EJB.

SampleServer-jar.xml Source File

The following line from the deployment descriptor file results from the transaction attribute in the definition in the eGen script.

```
<trans-attribute>Supports</trans-attribute>
```

As described in *Components of an eGen Server EJB Script* on page 3-6, this element indicates the level of transaction demarcation. If the transaction-attribute is not present in the definition, the default value is Supports. So, in this example, the transaction attribute was not listed in the script definition.

wl-SampleServer-jar.xml Source File

The following line from the WebLogic deployment information file results from the home interface name in the eGen script.

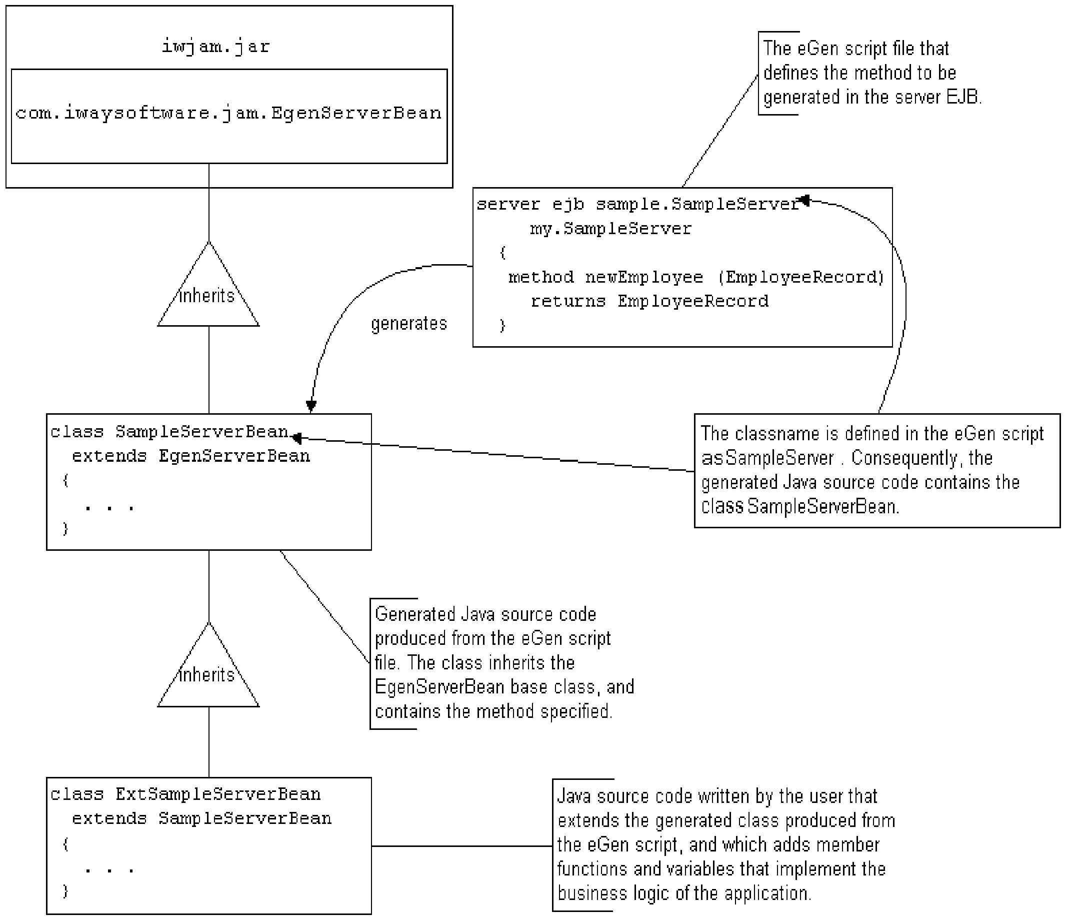
```
<jndi-name>my.sampleServer</jndi-name>
```

As described in *Components of an eGen Server EJB Script* on page 3-6, my.sampleServer is the home interface identifier for this bean in the WebLogic deployment description.

Customizing a Server Enterprise Java Bean-Based Application

The generated server enterprise Java bean-based applications are intended only for customizing, since they perform no real work without customizing. This topic describes the way generated server EJB code can be customized.

The following figure illustrates the Server EJB Class Hierarchy, the relationships and inheritance hierarchy between the iWay JAM classes comprising the application.


iwjam.jar
com.iwaysoftware.jam.EgenServerBean
inherits
The eGen script file that defines the method to be generated in the server EJB.
server ejb sample.SampleServer
my.SampleServer
{
method newEmployee (EmployeeRecord)
returns EmployeeRecord
}
generates
class SampleServerBean
extends EgenServerBean
{
. . .
}
The classname is defined in the eGen script asSampleServer . Consequently, the generated Java source code contains the class SampleServerBean.
inherits
Generated Java source code produced from the eGen script file. The class inherits the EgenServerBean base class, and contains the method specified.
class ExtSampleServerBean
extends SampleServerBean
{
. . .
}
Java source code written by the user that extends the generated class produced from the eGen script, and which adds member functions and variables that implement the business logic of the application.

The generated Java code for a server EJB application is a class that inherits the class EgenServerBean. The EgenServerBean class is provided in the iWay JAM distribution JAR file. This base class provides the basic framework for an EJB. It provides the required methods for a Stateless Session EJB.

The following sample, ExtSampleServerBean.java Contents, shows an example ExtSampleServerBean class that extends the generated SampleServerBean class, providing an implementation of the newEmployee() method. The example method prints a message indicating that a newEmployee request has been received.

```
package sample;

public class ExtSampleServerBean extends SampleServerBean
{
public EmployeeRecord newEmployee (EmployeeRecord in)
{
  System.out.println("New Employee: " +
+in.getEmpRecord().getEmpName().getEmpNameFirst()
+ " "
+ in.getEmpRecord().getEmpname().getEmpNameLast());
  return in;
}
}
```

After it is written, the ExtSampleServerBean class and the other EJB Java source files must be compiled and deployed in the same manner as other EJBs. Before deploying, the deployment descriptor must be modified; the ejb-class must be set to the name of your extended EJB implementation class. For more information, see *Deploying an iWay JAM eGen EJB* in Chapter 4, *Deploying Applications*.

For additional information about compiling and deploying, see the WebLogic Server documentation. The sample file provided with WebLogic Server contains a build script for reference.

WebLogic Server to Mainframe Application Models

In a WebLogic Server to Mainframe application, a request originates on a WebLogic client or server, and is serviced by a mainframe program invoked by the iWay JAM Gateway in cooperation with the CRM.

Generating a Stand-Alone Client Application

A stand-alone client application produces simple Java classes that perform the appropriate conversions of data records sent between Java and the mainframe and call mainframe services, but without all of the EJB support methods. These classes are intended to be lower-level components upon which more complicated applications are built.

Reference: Components of an eGen Stand-Alone Application Script

The general form of a definition of a stand-alone client class that appears in an eGen script is as follows (keywords are in bold):

```
client class classname
{ clientmethods }
```

The following table lists and describes the stand-alone client class definition keywords and parameters.

Keyword/Parameter	Definition
client class	Indicates to the eGen utility the type of application component that is to be generated.
classname	Indicates the class name for the client class. **Note**: The package name should be included in the class name.
clientmethods	List of methods that appear in the client class implementation (must be in braces). These methods are wrappers for calls to services that are defined in the services section of the eGen script. The general form of the definition for a client method in an eGen script is as follows: `method methodname is service servicename` The following table describes the parameters of a client method definition.

The following table lists and describes the parameters for the client method definition.

Parameter	Definition
methodname	Name of the method.
servicename	Indicates the remote service for which this method acts as a wrapper for an iWay JAM call. This service must be defined in the same eGen script.

The following is an example of a simple stand-alone client class definition that appears in an eGen script:

```
client class sample.SampleClass
{
method newEmployee
is service sampleCreate
}
```

where:

`sample`
: Is the package name

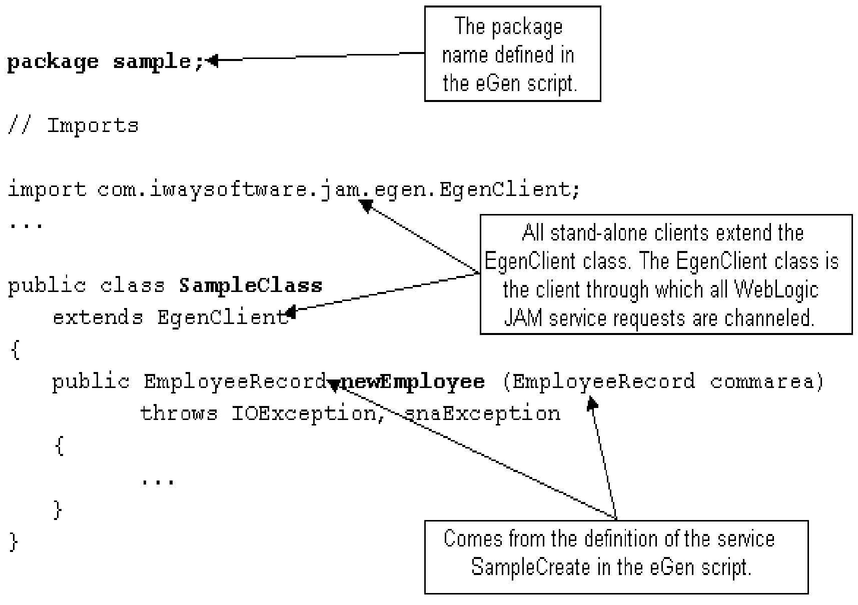
`SampleClass`
: Is the class name.

`newEmployee`
: Is the method that acts as a wrapper for an iWay JAM call to the remote service, sampleCreate.

`sampleCreate`
: Is the remote service that must be defined in the same eGen script as the client class.

The file SampleClass.java, containing the source for the sample class, is generated.

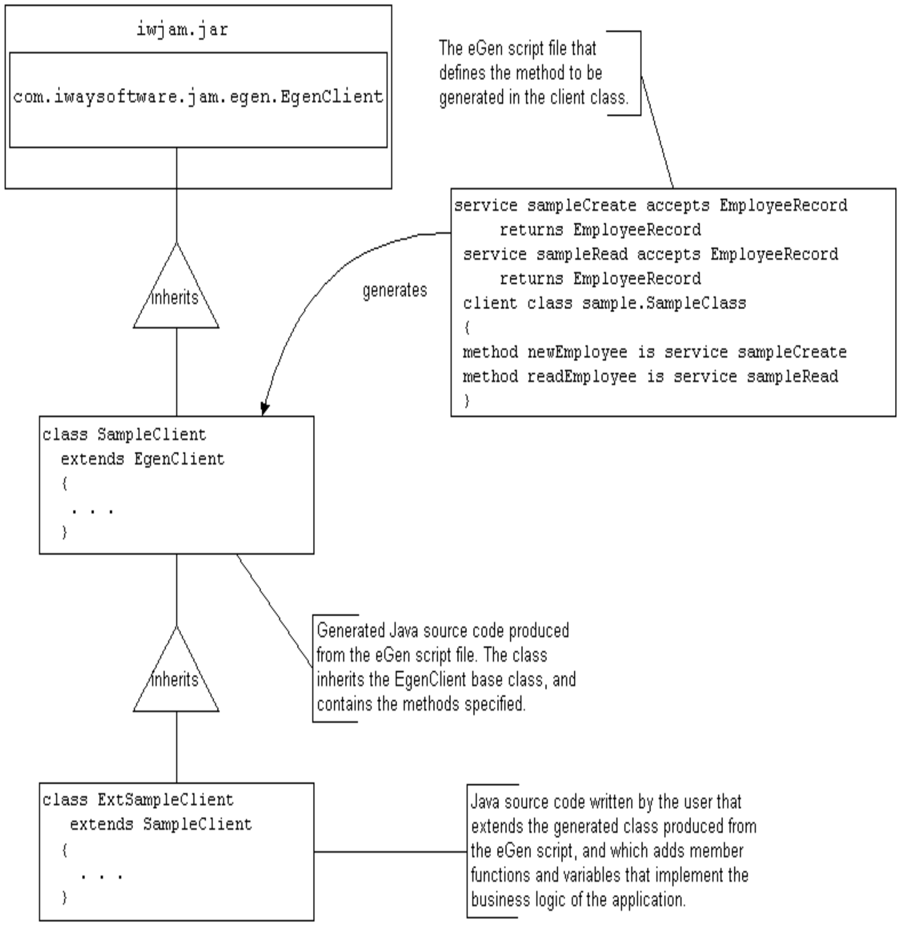
The following image shows the partial contents of the SampleClass.java source file.



```
package sample;

// Imports

import com.iwaysoftware.jam.egen.EgenClient;
...

public class SampleClass
    extends EgenClient
{
    public EmployeeRecord newEmployee (EmployeeRecord commarea)
           throws IOException, snaException
    {
           ...
    }
}
```

The package name defined in the eGen script.
All stand-alone clients extend the EgenClient class. The EgenClient class is the client through which all WebLogic JAM service requests are channeled.
Comes from the definition of the service SampleCreate in the eGen script.

The following figure illustrates customizing a stand-alone Java application, the iWay JAM client class hierarchy, and the relationships and inheritance hierarchy between the iWay JAM classes comprising the stand-alone java application.


iwjam.jar
com.iwaysoftware.jam.egen.EgenClient
The eGen script file that defines the method to be generated in the client class.
inherits
generates
inherits
Generated Java source code produced from the eGen script file. The class inherits the EgenClient base class, and contains the methods specified.
Java source code written by the user that extends the generated class produced from the eGen script, and which adds member functions and variables that implement the business logic of the application.

```
service sampleCreate accepts EmployeeRecord
    returns EmployeeRecord
 service sampleRead accepts EmployeeRecord
    returns EmployeeRecord
 client class sample.SampleClass
 {
 method newEmployee is service sampleCreate
 method readEmployee is service sampleRead
 }
```

```
class SampleClient
  extends EgenClient
  {
   . . .
  }
```

```
class ExtSampleClient
   extends SampleClient
  {
    . . .
  }
```

The generated Java code for a client class application is a class that inherits class EgenClient. The EgenClient class is provided in the iWay JAM distribution jam.jar file. This base class provides the basic framework for a client to the iWay JAM Gateway, as well as the required methods for accessing the gateway.

Your class, which extends or uses the SampleClient class, simply overrides or calls these methods to provide additional business logic, modifying the contents of the DataView. Your class may also add additional methods.

The following sample, ExtSampleClient.java Contents, shows an example ExtSampleClass class that extends the generated SampleClient class.

```
package sample;

public class ExtSampleClient extends SampleClass
{
// createEmployee
//
public EmployeeRecord newEmployee(EmployeeRecord
commarea)
throws IOException, snaException
{
  if (!isSsnValid(commarea.getEmpRecord().getEmpSsn()))
  {
  // The SSN is not valid
  throw new Error(”Invalid Social Security Number:”+
commarea.getEmpRecord().getEmpSsn());
  }
return super.newEmployee(commarea);
}
.
.
.
// Private functions

/*****************************************************
* Validates an SSN field.
*/

private boolean isSsnValid(BigDecimal ssn)
{
  if (ssn.longValue() < 100000000)
  {
    // Oops, appears to be less than 9 digits.
    return false;
  }
         return (true);
}
}
```

After it is written, the ExtSampleClient class and the other Java source files must be compiled and placed in your class path.

Instead of extending the generated client, you can also write classes that have the generated client as a member. This is an especially useful alternative if the class you write must extend another class.

Generating a Client Enterprise Java Bean-Based Application

A client enterprise Java bean-based application produces Java classes that comprise an EJB application. The class methods are invoked from requests originating from other EJB classes or other WebLogic Server client classes and transfer data records to and from the mainframe (remote system). From the viewpoint of the mainframe, the Java classes act as a remote CICS or IMS client. From the viewpoint of the WebLogic Server client classes, they act as regular EJB classes.

Reference: Components of an eGen Client EJB Script

To produce an EJB-based application, the script file that defines your DataViews must be edited to describe both the mainframe services accessed and the EJB that will access them.

The general form of a definition of a client (WebLogic Server to Mainframe) EJB that appears in an eGen script is as follows (keywords are in bold):

```
client ejb classname ejbregistration transaction
   transaction-attribute
{clientmethods}
```

The following table lists and describes the client EJB script keywords and parameters.

Keyword/Parameter	Definition
client ejb	Indicates to the eGen utility the type of application component that is to be generated.
classname	Indicates the class name stem for the EJB. For example, if the classname is SampleClient, the following files are generated by the eGen utility: • SampleClient.java • SampleClientBean.java • SampleClientHome.java **Note**: The package name should be included in the class name.
ejbregistration	Name to use to register the home interface for the EJB.

Keyword/Parameter	**Definition**
transaction transaction- attribute	This keyword and parameter are optional. They indicate the level of transaction demarcation. The possible values of transaction-attribute are: • NotSupported • Required • Supports • RequiresNew • Mandatory • Never **Note**: If the transaction keyword is not present in the definition, the default value of the transaction-attribute is Supports. For a detailed explanation of how the WebLogic Server EJB container responds to the transaction-attribute setting, see the section on Transaction Attributes in the EJB 2.0 Specification.
clientmethods	List of methods that appear in the EJB implementation. These methods are wrappers for calls to remote services that are defined in the services section of the eGen script. The general form of a client method definition is as follows (keywords are in bold): `method methodname is service servicename` The following table describes the parameters of a client method definition.

The following table describes the parameters of a client method definition.

Parameter	**Definition**
methodname	Name of the method.
servicename	Indicates the remote service for which this method acts as a wrapper for an iWay JAM call. This service must be defined in the same eGen script.

The following is an example of a client (WebLogic Server to Mainframe) EJB definition that appears in an eGen script:

```
client ejb sample.SampleClient my.sampleBean
{
method newEmployee
is service sampleCreate
}
```

where:

`sample`
: Is the package name.

`SampleClient`
: Is a Java bean class.

`newEmployee`
: Is the name of the method which corresponds to the service name.

`sampleCreate`
: Is the service name.

`my.sampleBean`
: Is the name under which the EJB home will be registered in JNDI (Java Naming and Directory Interface).

Reference: Generated Files

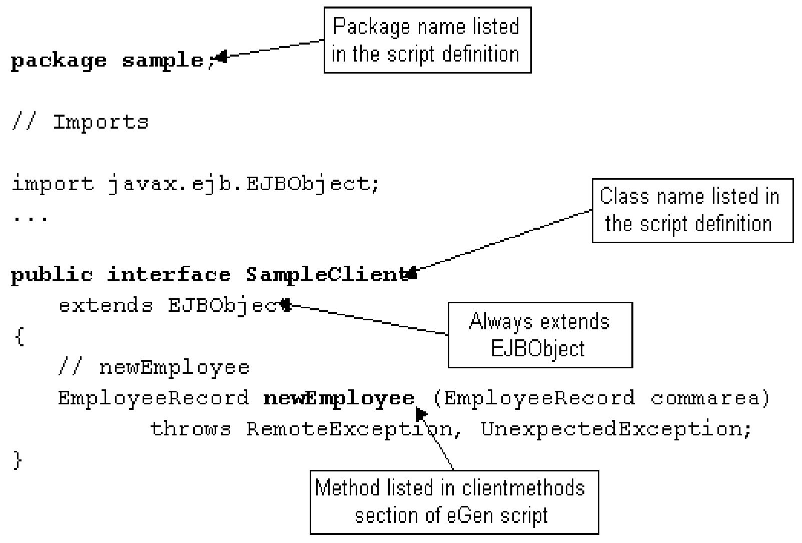
The following table lists the files generated from the client (WebLogic Server to Mainframe) EJB described in *Components of an eGen Client EJB Script* on page 3-18. These files are described in the topics following the table.

.

File	Content
SampleClient.java	Source for the EJB remote interface.
SampleClientBean.java	Source for the EJB implementation.
SampleClientHome.java	Source for the EJB home interface.
SampleClient-jar.xml	Deployment descriptor.
wl-SampleClient-jar.xml	WebLogic deployment information.

SampleClient.java Source File

The following figure shows the partial contents of the generated remote interface SampleClient.java source file. Following the example are descriptions of the elements in this file.

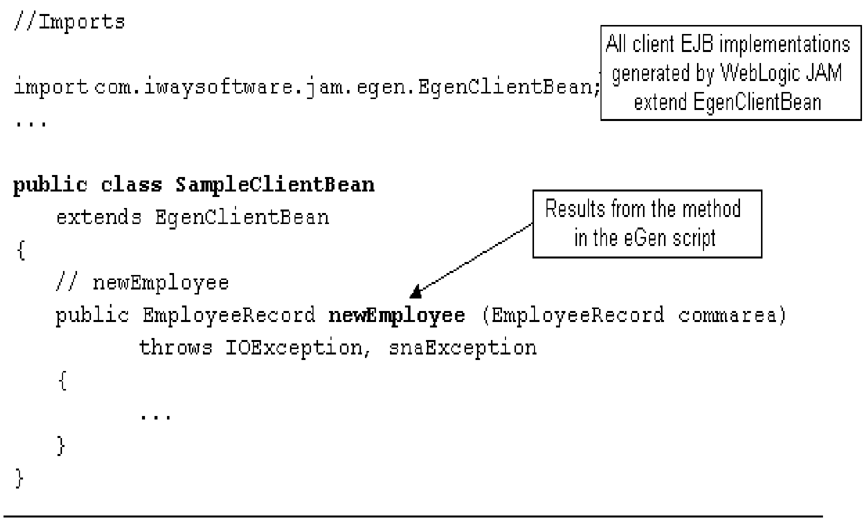


```
package sample;

// Imports

import javax.ejb.EJBObject;
...

public interface SampleClient
    extends EJBObject
{
    // newEmployee
    EmployeeRecord newEmployee (EmployeeRecord commarea)
            throws RemoteException, UnexpectedException;
}
```

Package name listed in the script definition
Class name listed in the script definition
Always extends EJBObject
Method listed in clientmethods section of eGen script

SampleClientBean.java Source File

The following figure shows the partial contents of the generated EJB implementation SampleClientBean.java source file. Following the example are descriptions of the elements in this file.



```
//Imports

import com.iwaysoftware.jam.egen.EgenClientBean;
...

public class SampleClientBean
    extends EgenClientBean
{
    // newEmployee
    public EmployeeRecord newEmployee (EmployeeRecord commarea)
            throws IOException, snaException
    {
            ...
    }
}
```

All client EJB implementations generated by WebLogic JAM extend EgenClientBean
Results from the method in the eGen script

SampleClientHome.java Source File

The eGen utility generates a standard home interface class for the client EJB.

SampleClient-jar.xml Source File

The following line from the deployment descriptor file results from the transaction demarcation listed in the definition in the eGen script.

```
<trans-attribute>Supports</trans-attribute>
```

As described in *Components of an eGen Client EJB Script* on page 3-18, this element indicates the level of transaction demarcation. If the transaction-attribute is not present in the definition, the default value is Supports. In this example, the transaction-attribute was not listed in the script definition.

wl-SampleServer-jar.xml Source File

The following line from the WebLogic deployment information file results from the Home Interface name in the eGen script.

```
<jndi-name>my.sampleBean</jndi-name>
```

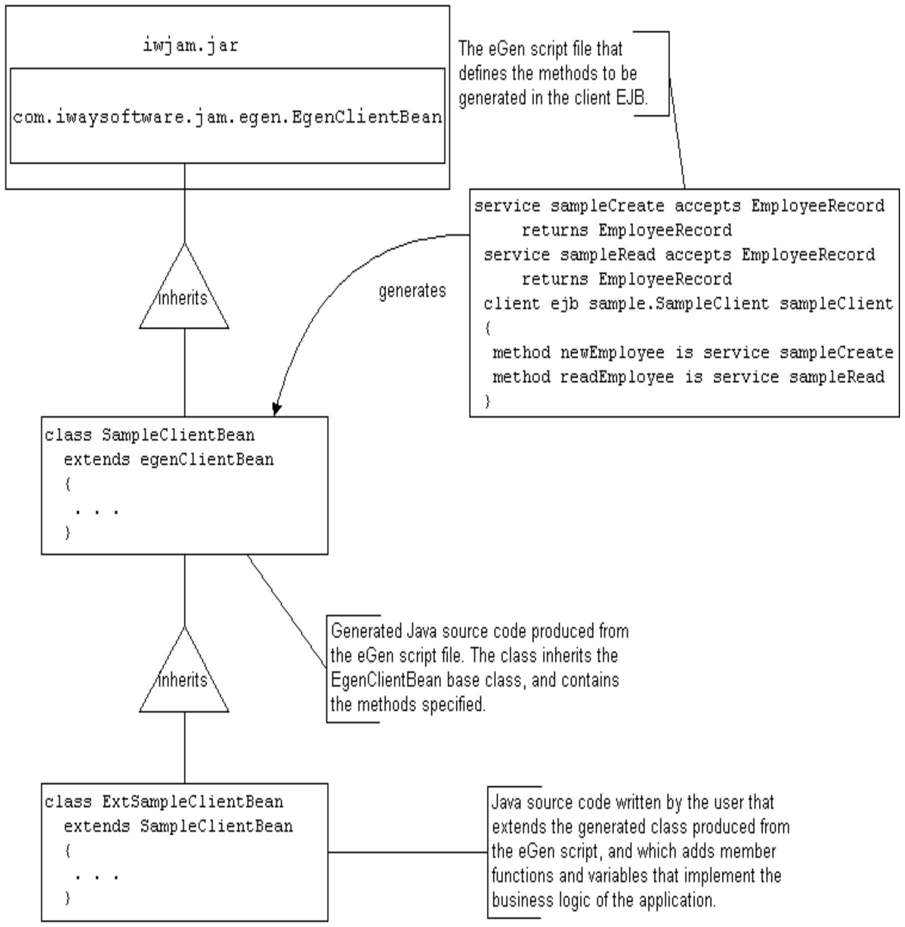
As described in *Components of an eGen Client EJB Script* on page 3-18, my.sampleBean is the home interface identifier for this bean in the WebLogic deployment description.

Note: You can edit the deployment descriptor to change the pool size, etc.

Customizing an Enterprise Java Bean-Based Application

The generated client enterprise Java bean-based applications are generally intended for customizing. Without customizing, the only function they perform is communication with the mainframe. This topic describes the way generated client EJB code can be customized.

The following figure illustrates the relationships and inheritance hierarchy between the iWay JAM classes comprising the application.


iwjam.jar
com.iwaysoftware.jam.egen.EgenClientBean
inherits
The eGen script file that defines the methods to be generated in the client EJB.
generates
class SampleClientBean extends egenClientBean { . . . }
Generated Java source code produced from the eGen script file. The class inherits the EgenClientBean base class, and contains the methods specified.
inherits
class ExtSampleClientBean extends SampleClientBean { . . . }
Java source code written by the user that extends the generated class produced from the eGen script, and which adds member functions and variables that implement the business logic of the application.

```
service sampleCreate accepts EmployeeRecord
     returns EmployeeRecord
 service sampleRead accepts EmployeeRecord
     returns EmployeeRecord
 client ejb sample.SampleClient sampleClient
 {
  method newEmployee is service sampleCreate
  method readEmployee is service sampleRead
 }
```

The generated Java code for a client EJB application is a class that inherits class egenClientBean. The egenClientBean class is provided in the iWay JAM distribution jar file.

The following sample illustrates an example ExtSampleClientBean class that extends the generated SampleClientBean class, adding a validation function (isSsnValid()) for the emp-ssn (m_empSsn) field of the DataView EmployeeRecord class. If the emp-ssn field is determined to be invalid, an exception occurs. Otherwise, the original function is called to perform the mainframe operation.

```
package Sample;

// Imports

import java.math.BigDecimal;
import java.io.IOException;

import com.iwaysoftware.jam.sna.jcrmgw.snaException;

// Local imports

import sample.EmployeeRecord;
import sample.SampleClientBean;

/***************************************************************
* Extends the SampleCientBean EJB class, adding additional business
logic.
*/

public class ExtSampleClientBean
extends SampleClientBean
{
//Public functions

...

/***************************************************************
 * Create a new employee record.
 */

 public EmployeeRecord newEmployee (EmployeeRecord commarea)
  throws IOException, snaException
{
if (!isSsnValid (commarea.getEmpRecord().getEmpSsn()))
{
// The SSN is not valid.
throw new Error ("Invalid Social Security Number:"
+ commarea.getEmpRecord().getEmpSsn());

}
//
  // Make the remote call.
  return super.newEmployee(commarea);
}
}
```

```
// Private Functions
/**********************************************************
* Validate an SSN field
*
* @return
* True if the SSN is valid, otherwise false.
*/

private boolean isSsnValid(final BigDecimal ssn)
{
if (ssn.longValue() < 100000000)
{
// Oops, appears to be less than 9 digits
return false;
}
return true;
}
}
```

After it is written, the ExtSampleClientBean class and the other EJB Java source files must be compiled and deployed in the same manner as other EJBs. Prior to deploying, the deployment descriptor must be modified; the ejb-class property must be set to the name of your extended EJB implementation class (see *Deploying an iWay JAM eGen EJB* in Chapter 4, *Deploying Applications*).

For more information about compiling and deploying, see the BEA WebLogic Server documentation. The sample file provided with WebLogic Server contains a build script for reference.

Generating a Servlet Application

An iWay JAM servlet application is a Java servlet that executes within BEA WebLogic Server. The application is started from a Web browser when the user enters a URL that is configured to invoke the servlet. The servlet presents an HTML form containing data fields and buttons. The buttons can be configured to invoke:

- EJB methods
- Remote gateway services (via the JAM Gateway)

In general, servlets generated by the eGen utility are intended for testing purposes and are not easily customized to provide a more aesthetically pleasing interface.

To produce a servlet application, create an eGen script file and use the eGen utility to generate your typed data record (DataView), and Servlet code.

In order to define a servlet application using an eGen script, you must define the following:

- HTML pages displayed by the servlet
- The servlet itself

Reference: Components of an eGen HTML Page Definition

The general form of an HTML page that appears in an eGen script is as follows (keywords are in bold):

```
page pagename title
{ view viewname
  buttons {buttonlist}
}
```

The following table lists and describes the HTML page definition keywords and parameters.

Keyword/Parameter	**Definition**
page	Indicates to the eGen utility the type of application component that is to be generated.
pagename	Indicates the name of the page so it can be referenced by the servlet and other page definitions in the script.
title	The title to be displayed on the HTML page.
viewname	Indicates the name of the DataView that to display on the page. This DataView must be defined elsewhere in the eGen script.
buttonlist	List of buttons that are displayed on the page. The buttons can either call EJB methods or remote services that are defined elsewhere in the eGen script. The general form of the definition for a button in the button list depends on whether it is a remote service button or an EJB.

The general syntax for a remote service button in an eGen script is as follows (keywords are in bold):

```
buttonname service (servicename) shows pagename
```

The following table lists and describes the remote service button definition keywords and parameters.

Keyword/Parameter	**Definition**
buttonname	Label that appears on the button.

Keyword/Parameter	Definition
servicename	Name of the remote service (must be in parenthesis).
pagename	Page used to display the results.

The general syntax for an EJB button in an eGen script is as follows (keywords are in bold):

```
buttonname ejbmethod () shows pagename
```

Note: Empty parenthesis must follow ejbmethod.

The following table lists and describes the EJB button definition keywords and parameters.

Keyword/Parameter	Definition
buttonname	Label that appears on the button.
ejbmethod	Name of the EJB method that is to be called. This method should be specified in the following form: `packagename.EJBclass.method`
pagename	Page used to display the results.

The following is an example of an HTML page that appears in an eGen script:

```
page initial "Initial Page"
{
view EmployeeRecord

buttons
{
"Create"
service ("sampleCreate")
shows fullPage
}
}
```

This listing defines an HTML page named initial, with a text title of Initial Page, that displays an EmployeeRecord record object as an HTML form. It also specifies that the form has a button labeled Create. When the button is pressed, the service sampleCreate is invoked and is passed the contents of the browser page as an EmployeeRecord object (the fields of which may have been modified by the user). Afterwards, the fullPage page is used to display the results.

Reference: Components of an eGen Servlet Definition

The general form of a servlet definition that appears in an eGen script is as follows (keywords are in bold):

```
servlet classname shows pagename
```

The following table lists and describes the servlet definition keywords and parameters.

Keyword/Parameter	Definition
servlet	Indicates the type of application component that is to be generated.
classname	Indicates the class name for the servlet. **Note**: The package name should be included in the class name.
pagename	Name of the page that is initially displayed by the servlet. This page must be defined elsewhere in the script.

Following is an example of the definition of an application servlet class that appears in an eGen script:

```
servlet sample.SampleServlet shows initial
```

where:

`sample`

Is the package name.

`SampleServlet`

Is an application servlet class.

The servlet is to be displayed in the HTML page named initial.

Generated Files

The eGen servlet definition described in *Components of an eGen Servlet Definition* on page 3-29 generates a servlet source code file called SampleServlet.java.

Customizing a Servlet iWay JAM Application

The generated Java classes produced for servlet applications are intended for proof of concept and prototypes. They can be customized in limited ways. It is presumed that some other development tool will be used to develop a servlet or other user interface on top of the generated EJBs or client classes.

Supplying Security Credentials

iWay JAM has the capability to accept user ID and password information from a Java client program, and apply that information to access a secure service on the mainframe.

Note: When security information is transmitted via the connection between the iWay JAM Gateway and the CRM, it is sent in clear text (not encrypted). You should not send this information over a network that can be read by unauthorized parties.

The following three levels of security are supported by iWay JAM.

- **Local.** No user information from the Java client is required to access a mainframe service. Use of this security level implies that any user with access to execute the Java client program should have access to a mainframe service.
- **Identify.** A user ID specified by the Java client is required to access a mainframe service. This user ID is passed to the mainframe to verify that it is a valid user ID. Use of this security level implies that there is a trusted relationship between the Java and mainframe environments, since there is no re-verification of the user's identity in the mainframe environment.
- **Verify.** A user ID and password specified by the Java client are required to access a mainframe service. The password is used to re-verify the user's identity in the mainframe environment.

For information on setting the security level for a CRM link and using a default user ID, see the *iWay Java Adapter for Mainframe Configuration and Administration Guide*.

For more information about establishing and administrating mainframe security, see your mainframe security documentation.

Supplying Security Credentials in a iWay JAM Client Program

User security information can be supplied in a iWay JAM stand-alone client or client EJB. The following methods in the EgenClient object support this operation.

`EgenClient.setUserId(String)`

This method sets the user ID to the value specified in the String argument.

`EgenClient.setPassword(String)`

This method sets the user password to the value specified in the String argument.

These methods can be called on any sub-class of EgenClient, such as the client classes generated by the eGen utility. The methods are not inserted automatically by the eGen utility. They must be manually added to the client program source and should be called prior to the calls to EgenClient.callService().

The methods, setUserID and setPassword, can be called on any subclass of EgenClientBean, such as the client EJBs generated by the eGen utility. EgenClientBean has methods by the same name that act as wrappers for calls to methods of the EgenClient member of the EgenClientBean class.

Calls to the EgenClient.setUserId() method within a iWay JAM client override any default user ID value configured for the CRM link the client is using.

These methods cannot be used with the servlet-only applications, since they do not use the EgenClient object directly. Servlet-only applications can make use of the default user ID to support the security level, Identify.

The following example illustrates a class that extends the generated EJB implementation to provide security credentials to the Gateway during these operations.

```
// ExtSampleClientBean.java
//

package sample;

// Imports
//
import java.io.IOException;
import com.iwaysoftware.jam.sna.jcrmgw.snaException;

/**
 * EJB implementation.
 */
public class ExtSampleClientBean extends SampleClientBean
{
    protected byte[] callService(String svc, byte[] input)
            throws snaException, IOException
    {
            setUserid("JAMUSER");
            setPassword("JAMPASS");

            return super.callService(svc, input);
    }
}

// END ExtSampleClientBean.java
```

Note: iWay JAM returns an SNANotAuthorized exception if the credentials are rejected by the mainframe security package.

iWay JAM to JMS

iWay JAM includes an EJB that has the following main functions:

- Inserts request data into JMS topics or queues.
- Converts EBCDIC data into an ASCII XML document for use with custom applications.

iWay JAM to JMS is a utility stateless session EJB that uses a DataView generated by the eGen utility to convert the data. The EJB is contained in the jam.ear file with a default JNDI name of JAMToJMS.

The general process for this insertion and conversion is described in the following steps.

1. Obtain a COBOL Copybook.

   The mainframe client application must have a COBOL record layout (copybook) to describe the message comprising the request. This layout is used to generate Java classes that can be used for data transformation. For more information, see *Working With COBOL Copybooks* in Chapter 2, *Generating a Java Application with eGen*.

2. Generate a DataView with XML Support.

   Ensure that your eGen script is written to generate DataViews that support XML, as shown in the following code example:

   ```
   generate view empRecData from emprec support xml
   ```

   For more information on DataViews, see *Writing the DataView Section of an eGen Script* in Chapter 2, *Generating a Java Application with eGen*.

   For more information on generating the DataView source files, see *Processing eGen Scripts with the eGen Utility* in Chapter 2, *Generating a Java Application with eGen*.

   These files can be compiled for deployment. The schema and DTD can be made available to the XML application as necessary.

3. Compile the DataView JAVA files (see *Creating an Environment for Generating and Compiling the Java Code* in Chapter 2, *Generating a Java Application with eGen*).

4. Copy the DataView class files created by the eGen utility to a directory in the WebLogic Server class path.

5. Create a JMS Event definition. For specific instructions, refer to the *BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide.*

For an example of how to use the iWay JAM to JMS feature, see the *BEA WebLogic Java Adapter for Mainframe Samples Guide*.

CHAPTER 4

Deploying Applications

Topics:

Deployment is the process of installing servlets and/or EJBs on WebLogic Server. Application deployment in WebLogic Server has evolved to the J2EE standard for web application deployment.

The following information is not intended to specifically describe how applications are deployed in WebLogic Server.

Deploying an iWay JAM eGen EJB

An iWay JAM eGen EJB (client or server) is deployed like any other WebLogic EJB. Considerations that are specific to iWay JAM are:

- Deployment descriptors generated by the eGen utility need to be renamed.
- If the EJB is to contain business logic in addition to iWay JAM access code, a subclass must be created.
- If multiple EJBs are created, the generated deployment descriptors must be manually merged if the beans are to be deployed in the same JAR file.

Renaming Deployment Descriptors

The EJB deployment descriptors generated by the eGen utility are named based on the generated EJB, rather than the using the standard J2EE and WebLogic file names. This is to avoid file naming conflicts if multiple beans are generated in the same directory. As a result, these descriptors must be renamed before the EJB is packaged and deployed. Following are the naming conventions used, where BeanName is the name of the generated EJB:

Generated Descriptor Name	Deployed Descriptor Name
`BeanName-jar.xml`	`ejb-jar.xml`
`wl-BeanName.xml`	`weblogic-jar.xml`

For example, consider the following portion of an eGen script:

```
client ejb TestClient TestClientHome
{
method newEmployee
is service emplCreate
}
```

In this script, the descriptions generated would be named `TestClient-jar.xml` and `wl-TestClient.xml`, respectively.

Adding Business Logic to a Generated EJB

The EJBs generated by the eGen utility contain the infrastructure for calling mainframe services and returning the results of those services. If you want to present a different API that performs some business logic before deferring to the generated service methods, you will need to create a new bean class that sub-classes the generated code.

If you want to maintain the same remote interface generated by the eGen utility but add business logic before/after the mainframe call, simply derive a new class from the generated bean class while retaining the generated home and remote interfaces. For example, if our generated `TestClientBean.java` contains a method named newEmployee(), you could insert business logic as follows:

```
public class MyLogicBean extends TestClientBean
{
public dataView newEmployee(dataView in)
{
// perform before business logic here
dataView out = super.newEmployee(in);
// perform after business logic here
return(out);
}
}
```

However, if you want to present a different remote interface in addition to adding business logic, you also need to create new remote and home interfaces to support your new bean.

In either case, be sure to update the generated deployment descriptors to reflect your new bean classes.

For example, suppose you used the eGen utility to generate an EJB named TestClientBean, and that bean had been extended as in the above example by a bean class named MyLogicBean. The eGen utility would have generated a deployment descriptor with the name `TestClient-jar.xml`. The generated deployment descriptor would need to be renamed `ejb-jar.xml` before deployment. The EJB-CLASS element's value should also be changed from TestClientBean to MyLogicBean to reflect the new bean class name as in the example below.

```
<ejb-jar>
    <enterprise-beans>
      <session>
        <ejb-name>TestClient</ejb-name>
        <home>TestClientHome</home>
        <remote>TestClient</remote>
        <ejb-class>MyLogicBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      </session>
    </enterprise-beans>
...
</ejb-jar>
```

Merging Multiple Deployment Descriptors

Multiple iWay JAM EJBs can be generated as part of a single application. This can be done in a single eGen script, or by running the eGen utility multiple times with different scripts. If these beans are to be deployed in a single JAR file, the generated deployment descriptors for each must be merged into a single ejb-jar.xml and weblogic-jar.xml.

Sample EJB Deployment

Following are instructions for the deployment of a sample eGen-created EJB.

1. Build your EJB deployment JAR file.
2. Deploy the EJB in BEA WebLogic Server by configuring it as a new EJB in the WebLogic Administration Console.

CHAPTER 5

Understanding Programming Flows

Topics:

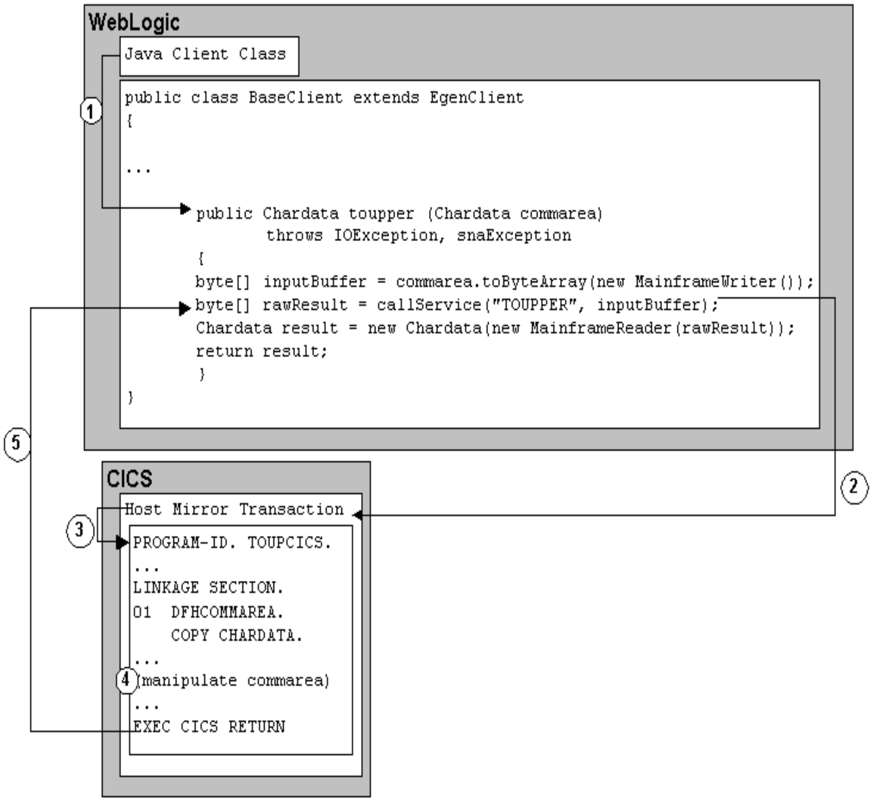
This section describes the interaction between WebLogic Server and mainframe programs.

Distributed Program Link Programming Flows

The following topics discuss and illustrate DPL programming flows.

Java Client Request/Response to CICS DPL

The following figure shows a Java Client Request/Response to CICS DPL programming flow.



```
public class BaseClient extends EgenClient
{

...

        public Chardata toupper (Chardata commarea)
               throws IOException, snaException
        {
        byte[] inputBuffer = commarea.toByteArray(new MainframeWriter());
        byte[] rawResult = callService("TOUPPER", inputBuffer);
        Chardata result = new Chardata(new MainframeReader(rawResult));
        return result;
        }
}
```

```
PROGRAM-ID. TOUPCICS.
...
LINKAGE SECTION.
01  DFHCOMMAREA.
    COPY CHARDATA.
...
(manipulate commarea)
...
EXEC CICS RETURN
```

The following steps describe the Java Client Request/Response to CICS DPL programming flow.

1. A Java client class, for example, a stand-alone client, EJB, makes a call to the BaseClient.toupper method with a Chardata DataView as the parameter.
2. In the toupper method, a call is made to the EgenClient.callService method.

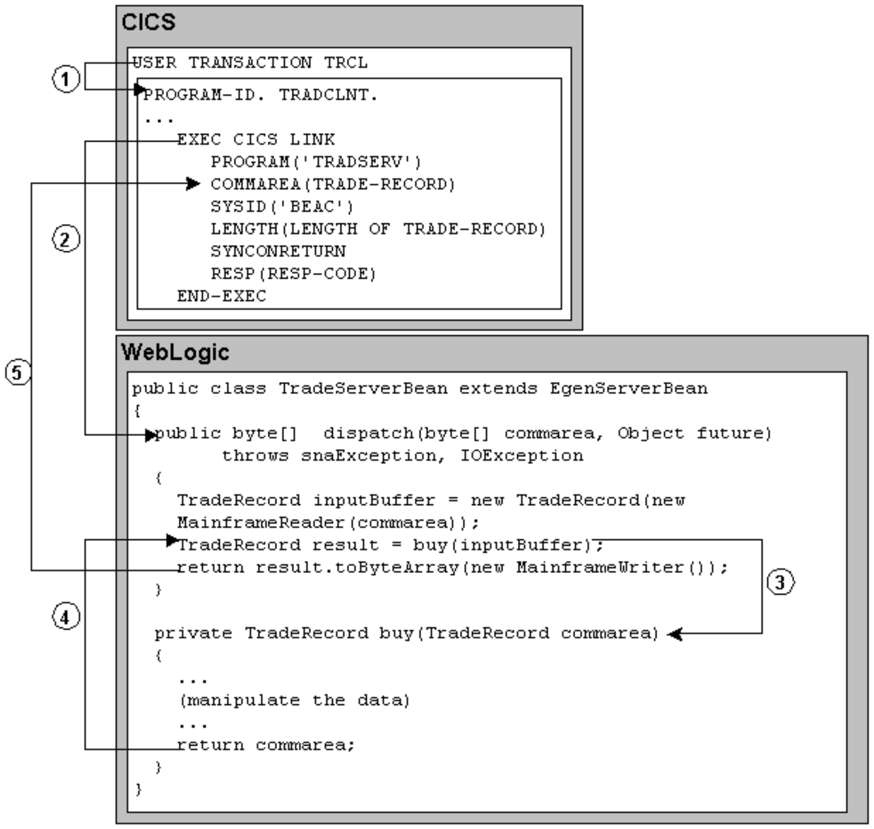
   Note: The BaseClient extends EgenClient, so the BaseClient inherits the callService method from EgenClient.

The value of the first parameter is TOUPPER. TOUPPER is the name of the DPL Service that is mapped to the CICS DPL program TOUPCICS in the WebLogic Administrative Console.

3. The host mirror transaction starts the TOUPCICS program and passes the contents of the inputBuffer byte array as the commarea.
4. The TOUPCICS program processes the data.
5. The CICS server returns the commarea. The data is returned from the EgenClient.callService method as the byte array rawResult.

CICS Request/Response DPL to WebLogic Server EJB

The following figure shows a CICS request/response DPL to WebLogic Server EJB programming flow.



```
USER TRANSACTION TRCL
 PROGRAM-ID. TRADCLNT.
 ...
    EXEC CICS LINK
       PROGRAM('TRADSERV')
       COMMAREA(TRADE-RECORD)
       SYSID('BEAC')
       LENGTH(LENGTH OF TRADE-RECORD)
       SYNCONRETURN
       RESP(RESP-CODE)
    END-EXEC
```

```
public class TradeServerBean extends EgenServerBean
{
  public byte[]  dispatch(byte[] commarea, Object future)
        throws snaException, IOException
  {
    TradeRecord inputBuffer = new TradeRecord(new
    MainframeReader(commarea));
    TradeRecord result = buy(inputBuffer);
    return result.toByteArray(new MainframeWriter());
  }

  private TradeRecord buy(TradeRecord commarea)
  {
    ...
    (manipulate the data)
    ...
    return commarea;
  }
}
```

The following steps describe the CICS request/response DPL to WebLogic Server EJB programming flow.

1. The user-entered transaction TRCL invokes the TRADCLNT program.

   The EXEC CICS LINK command causes the advertised service TRADSERV to execute. The SYSID value is set to the name of the connection associated with the CRM Logical Unit. The SYNCONTRETURN parameter indicates that the WebLogic Server EJB will not be involved in the CICS transaction.

2. In the WebLogic Administration Console, the TRADSERV service is mapped to the JNDI name, jam.TradeServer, for the TradeServer EJB. This causes the dispatch method of TradeServerBean to be invoked.

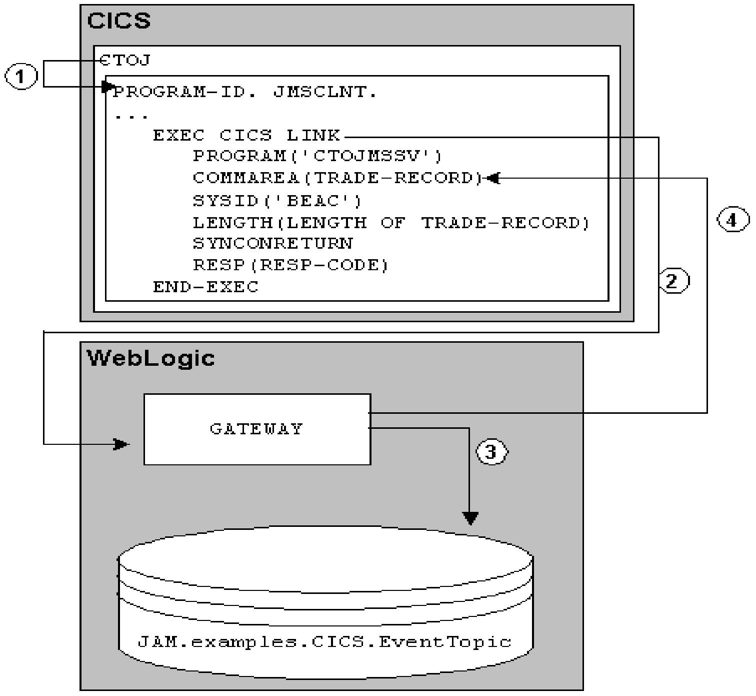
3. The buy method is invoked from the dispatch method.

4. The business logic is performed, and the result is returned to the dispatch method.

5. The data is returned from the dispatch method into the COMMAREA.

CICS DPL Asynchronous No Reply to WebLogic Server Application

The following figure shows a CICS DPL asynchronous no reply to Java server programming flow.



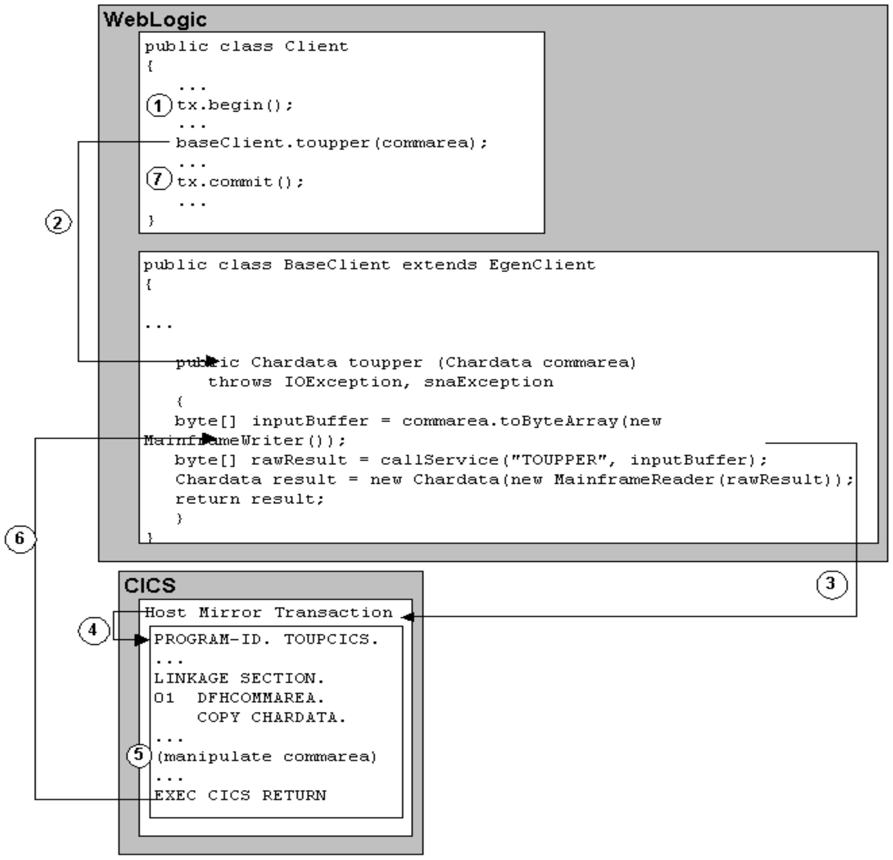
```
PROGRAM-ID. JMSCLNT.
...
   EXEC CICS LINK
      PROGRAM('CTOJMSSV')
      COMMAREA(TRADE-RECORD)
      SYSID('BEAC')
      LENGTH(LENGTH OF TRADE-RECORD)
      SYNCONRETURN
      RESP(RESP-CODE)
   END-EXEC
```

The following steps describe the CICS DPL asynchronous no reply to Java server programming flow.

1. The user-entered transaction CTOJ invokes the JMSCLNT program.
2. The EXEC CICS LINK command causes the advertised service CTOJMSSV to execute. The SYSID value is set to the name of the connection associated with the CRM Logical Unit. The SYNCONTRETURN parameter indicates that the WebLogic Server EJB will not be involved in the CICS transaction.
3. The Gateway sends the message to the JMS Event CTOJMSSV.
4. In the WebLogic Administration Console, the CTOJMSSV service name is mapped to the JMS topic Jam.examples.CICS.EventTopic.
5. Data that is identical to the request data is returned in the COMMAREA to JMSCLNT.

Transactional Java Client Request/Response to CICS DPL

The following figure shows a transactional Java client request/response to CICS DPL programming flow.



```
public class Client
{
    ...
    tx.begin();
    ...
    baseClient.toupper(commarea);
    ...
    tx.commit();
    ...
}
```

```
public class BaseClient extends EgenClient
{

...

   public Chardata toupper (Chardata commarea)
      throws IOException, snaException
   {
   byte[] inputBuffer = commarea.toByteArray(new
MainframeWriter());
   byte[] rawResult = callService("TOUPPER", inputBuffer);
   Chardata result = new Chardata(new MainframeReader(rawResult));
   return result;
   }
}
```

```
PROGRAM-ID. TOUPCICS.
...
LINKAGE SECTION.
01  DFHCOMMAREA.
    COPY CHARDATA.
...
(manipulate commarea)
...
EXEC CICS RETURN
```

The following steps describe the transactional Java client request/response to CICS DPL programming flow.

1. A Java client class calls the begin method of a UserTransaction object to start a transaction.
2. Within the boundaries of that transaction, the Java client class makes a call to the BaseClient.toupper method with a Chardata DataView as the parameter.
3. In the toupper method, a call is made to the EgenClient.callService method.

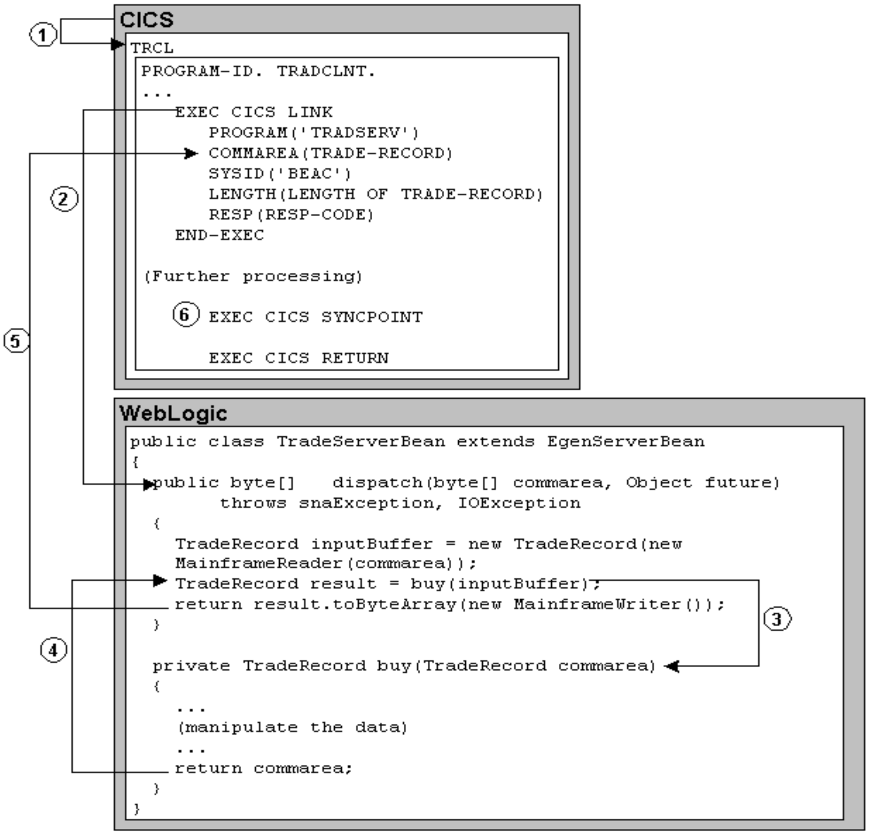
Note: The BaseClient extends EgenClient, so the BaseClient inherits the callService method from EgenClient.

The value of the first parameter is TOUPPER. TOUPPER is the name of the DPL Service that is mapped to the CICS DPL program TOUPCICS in the WebLogic Administration Console.

4. The host mirror transaction starts the TOUPCICS program and passes the contents of the inputBuffer byte array as the commarea.
5. The TOUPCICS program processes the data.
6. The CICS server returns the commarea. The data is returned from the EgenClient.callService method as the byte array rawResult.
7. The Java client class calls the commit method of the UserTransaction object to indicate the successful completion of the transaction. This causes the commit of the WebLogic managed resources, as well as the resources held by the Host Mirror Transaction.

Transactional CICS Request/Response DPL to WebLogic Server EJB

The following figure shows a transactional CICS request/response DPL to WebLogic Server EJB programming flow.



```
TRCL
PROGRAM-ID. TRADCLNT.
...
   EXEC CICS LINK
      PROGRAM('TRADSERV')
      COMMAREA(TRADE-RECORD)
      SYSID('BEAC')
      LENGTH(LENGTH OF TRADE-RECORD)
      RESP(RESP-CODE)
   END-EXEC

(Further processing)

      EXEC CICS SYNCPOINT

      EXEC CICS RETURN
```

```
public class TradeServerBean extends EgenServerBean
{
  public byte[]   dispatch(byte[] commarea, Object future)
        throws snaException, IOException
  {
    TradeRecord inputBuffer = new TradeRecord(new
    MainframeReader(commarea));
    TradeRecord result = buy(inputBuffer);
    return result.toByteArray(new MainframeWriter());
  }

  private TradeRecord buy(TradeRecord commarea)
  {
    ...
    (manipulate the data)
    ...
    return commarea;
  }
}
```

The following steps describe the transactional CICS request/response DPL to WebLogic Server EJB programming flow.

1. The user-entered transaction TRCL invokes the TRADCLNT program.
2. The EXEC CICS LINK command causes the advertised service TRADSERV to execute. The SYSID value is set to the name of the connection associated with the CRM Logical Unit. When the SYNCONRETURN command is not included in the EXEC CICS LINK, this indicates that the WebLogic Server is involved in the CICS transaction.

In the WebLogic Administration Console, the TRADSERV service is mapped to the JNDI name, jam.TradeServer, for the TradeServer EJB. This causes the dispatch method of TradeServerBean to be invoked.

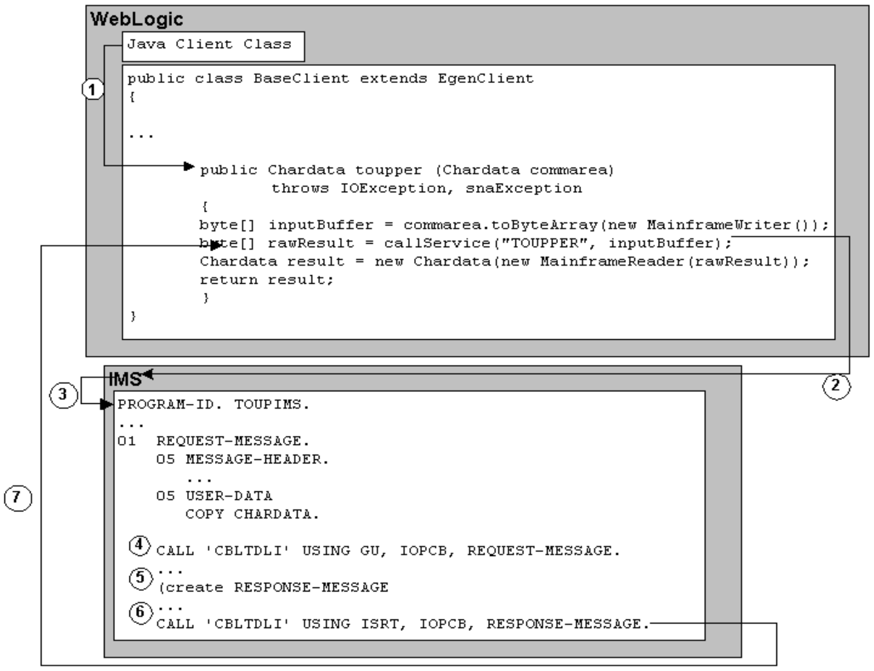
3. The buy method is invoked from the dispatch method.
4. The business logic is performed, and the result is returned to the dispatch method.
5. The data is returned from the dispatch method into the COMMAREA.
6. If necessary, further processing can be done in TRADCLNT before the EXEC CICS SYNCPOINT that ends the transaction.

IMS Implicit APPC Programming Flows

The following topics discuss and illustrate IMS implicit APPC programming flows.

Java Client Request/Response to IMS Transaction Program

The following figure shows a Java Client Request/Response to IMS transaction programming flow.



```
public class BaseClient extends EgenClient
{

...

        public Chardata toupper (Chardata commarea)
                throws IOException, snaException
        {
        byte[] inputBuffer = commarea.toByteArray(new MainframeWriter());
        byte[] rawResult = callService("TOUPPER", inputBuffer);
        Chardata result = new Chardata(new MainframeReader(rawResult));
        return result;
        }
}
```

```
PROGRAM-ID. TOUPIMS.
...
01  REQUEST-MESSAGE.
    05 MESSAGE-HEADER.
       ...
    05 USER-DATA
       COPY CHARDATA.

    CALL 'CBLTDLI' USING GU, IOPCB, REQUEST-MESSAGE.
    ...
    (create RESPONSE-MESSAGE
    ...
    CALL 'CBLTDLI' USING ISRT, IOPCB, RESPONSE-MESSAGE.
```

The following steps describe the Java Client Request/Response to IMS programming flow.

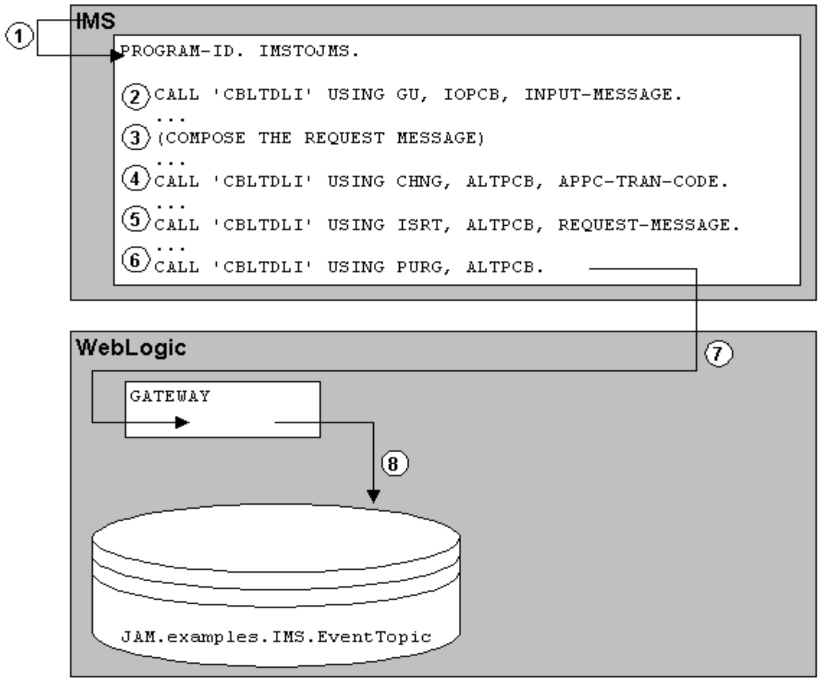
1. A Java client class (such as a stand-alone client, EJB, etc.) makes a call to the BaseClient.toupper method with a Chardata DataView as the parameter.
2. In the toupper method, a call is made to the EgenClient.callService method.

   Note: The BaseClient extends EgenClient, so the BaseClient inherits the callService method from EgenClient.

   The value of the first parameter is TOUPPER. TOUPPER is the name of the APPC Service that is mapped to the IMS transaction TOUPIMS in the WebLogic Administrative Console.
3. IMS starts the TOUPIMS transaction. This transaction executes the associated program TOUPIMS. The contents of the inputBuffer byte array are placed on an IOPCB as request data.
4. The TOUPIMS program accesses the request data by performing a get unique on the IOPCB.
5. The TOUPIMS program processes the data and creates a response message.
6. The TOUPIMS program inserts the response data to the IOPCB.
7. IMS returns the response data back to the requester. The data is returned from the EgenClient.callService method as the byte array rawResult.

IMS Asynchronous No Reply Transaction Program to Java Server

The following figure shows an IMS asynchronous no reply transaction program to a Java server programming flow.



```
PROGRAM-ID. IMSTOJMS.
CALL 'CBLTDLI' USING GU, IOPCB, INPUT-MESSAGE.
...
(COMPOSE THE REQUEST MESSAGE)
...
CALL 'CBLTDLI' USING CHNG, ALTPCB, APPC-TRAN-CODE.
...
CALL 'CBLTDLI' USING ISRT, ALTPCB, REQUEST-MESSAGE.
...
CALL 'CBLTDLI' USING PURG, ALTPCB.
```

The following steps describe the IMS transaction program to asynchronous no reply Java Server programming flow.

1. IMS starts the IMSTOJMS transaction. This transaction executes the associated program IMSTOJMS.
2. The IMSTOJMS program accesses the input data by doing a get unique on the IOPCB.
3. The IMSTOJMS program composes the request message.
4. The IMSTOJMS program issues a call with the CHNG function code to store the appropriate logical terminal name in a modifiable alternate PCB.

   Note: To use an alternate PCB, you must include a PCB statement in your PSB, for example, IMS PSBGEN for a Modifiable Alternate PCB for the IMS Client.

   ```
   PCB    TYPE=TP,MODIFY=YES
   PSBGEN PSBNAME=IMSTOJMS,CMPAT=YES,LANG=COBOL
   ```

Note: The logical terminal name, in this case, JAMIMS01, must be mapped to an LU name and a transaction name in a LU 6.2 Descriptor. In the following example, JAMIMS01 is mapped to the LU CRMLU and the transaction ITOJMSSV.

```
A JAMIMS01 LUNAME=CRMLU TPNAME=ITOJMSSV SYNCLEVEL=N
```

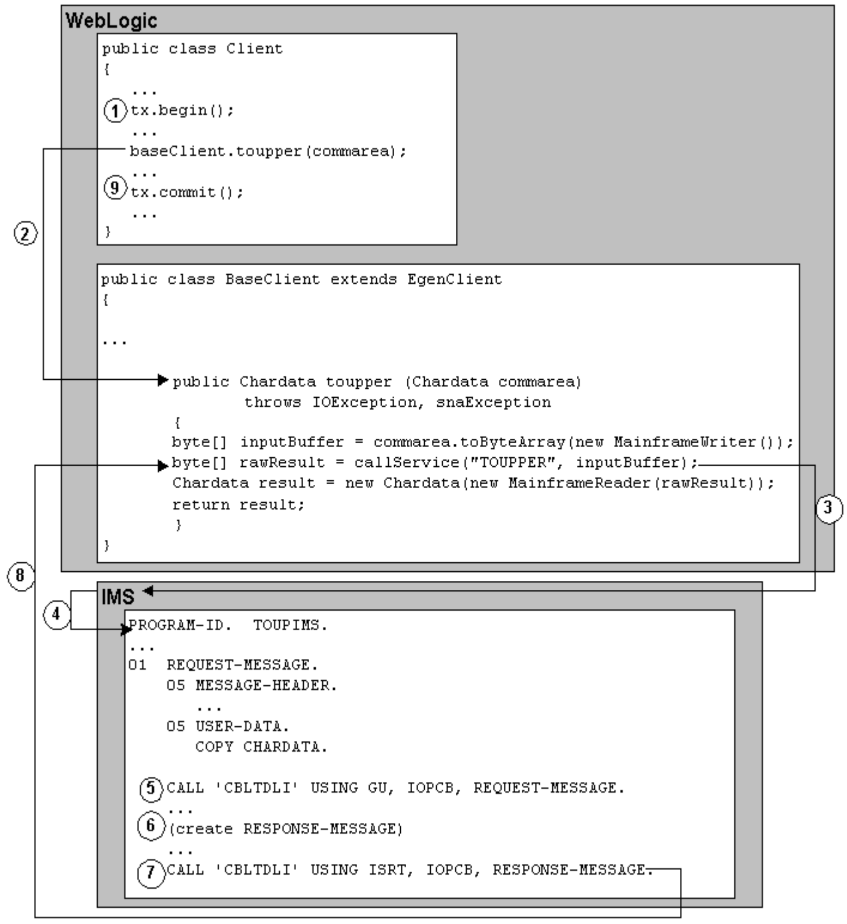
5. The IMSTOJMS program issues an insert call with the request message to the alternate PCB, ALTPCB.
6. The IMSTOJMS program issues a PURG call to the alternate PCB, ALTPCB, to tell IMS to send the request message.
7. IMS sends the request message to the indicated LU, the LU configured for the CRM. The request message is forwarded to the Gateway.
8. The gateway sends the message to the JMS Event ITOJMSSV. ITOJMSSV is the transaction name in the previous, LU 6.2 descriptor:

   ```
   A JAMIMS01 LUNAME=CRMLU TPNAME=ITOJMSSV SYNCLEVEL=N
   ```

   In the WebLogic Administration Console, the ITOJMSSV service name is mapped to the JMS topic JAM.examples.IMS.EventTopic.

Transactional Java Client Request/Response to IMS Transaction Program

The following figure shows a transactional Java client request/response to an IMS transaction programming flow.



```
public class Client
{
   ...
   tx.begin();
   ...
   baseClient.toupper(commarea);
   ...
   tx.commit();
   ...
}
```

```
public class BaseClient extends EgenClient
{

...

          public Chardata toupper (Chardata commarea)
                  throws IOException, snaException
           {
          byte[] inputBuffer = commarea.toByteArray(new MainframeWriter());
          byte[] rawResult = callService("TOUPPER", inputBuffer);
          Chardata result = new Chardata(new MainframeReader(rawResult));
          return result;
           }
}
```

```
PROGRAM-ID.  TOUPIMS.
...
01  REQUEST-MESSAGE.
    05 MESSAGE-HEADER.
       ...
    05 USER-DATA.
       COPY CHARDATA.

    CALL 'CBLTDLI' USING GU, IOPCB, REQUEST-MESSAGE.
    ...
    (create RESPONSE-MESSAGE)
    ...
    CALL 'CBLTDLI' USING ISRT, IOPCB, RESPONSE-MESSAGE.
```

The following steps describe the transactional Java client request/response to IMS transaction programming flow.

1. A Java client class calls the begin method of a UserTransaction object to start a transaction.
2. Within the boundaries of that transaction, the Java client class makes a call to the BaseClient.toupper method with a Chardata DataView as the parameter.

3. In the toupper method, a call is made to the EgenClient.callService method.

   Note: The BaseClient extends EgenClient, so the BaseClient inherits the callService method from EgenClient.

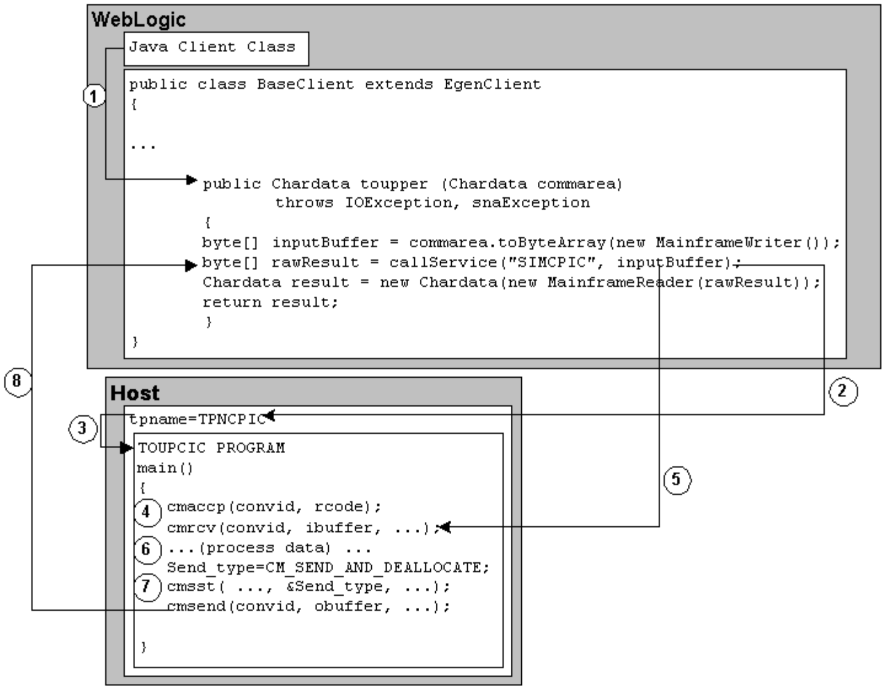
   The value of the first parameter is TOUPPER. TOUPPER is the name of the APPC Service that is mapped to the IMS transaction TOUPIMS in the WebLogic Administration Console.
4. IMS starts the TOUPIMS transaction that executes the associated program TOUPIMS. The contents of the inputBuffer byte array are placed on an IOPCB as request data.
5. The TOUPIMS program accesses the request data by doing a get unique on the IOPCB.
6. The TOUPIMS program processes the data and creates a response message.
7. The TOUPIMS program inserts the response data to the IOPCB.
8. IMS returns the response data back to the requester. The data is returned from the EgenClient.callService method as the byte array rawResult.
9. The Java client class calls the commit method of the UserTransaction object to indicate the successful completion of the transaction. This causes the commit of the WebLogic managed resources, as well as the resources managed by IMS.

Common Programming Interface for Communications Programming Flows

The following topics discuss and illustrate CPI-C programming flows.

Java Client Request/Response to Host CPI-C

The following figure shows a Java client request/response to a host CPI-C programming flow.



```
public class BaseClient extends EgenClient
{

...

        public Chardata toupper (Chardata commarea)
               throws IOException, snaException
        {
        byte[] inputBuffer = commarea.toByteArray(new MainframeWriter());
        byte[] rawResult = callService("SIMCPIC", inputBuffer);
        Chardata result = new Chardata(new MainframeReader(rawResult));
        return result;
        }
}
```

```
tpname=TPNCPIC
TOUPCIC PROGRAM
main()
{
   cmaccp(convid, rcode);
   cmrcv(convid, ibuffer, ...);
   ...(process data) ...
   Send_type=CM_SEND_AND_DEALLOCATE;
   cmsst( ..., &Send_type, ...);
   cmsend(convid, obuffer, ...);

}
```

The following steps describe the Java client request/response to host CPI-C programming flow.

1. A Java client class (such as a stand-alone client, EJB, etc.) makes a call to the BaseClient.toupper method with a Chardata dataview as the parameter.
2. In the toupper method, a call is made to the EgenClient.callService method.

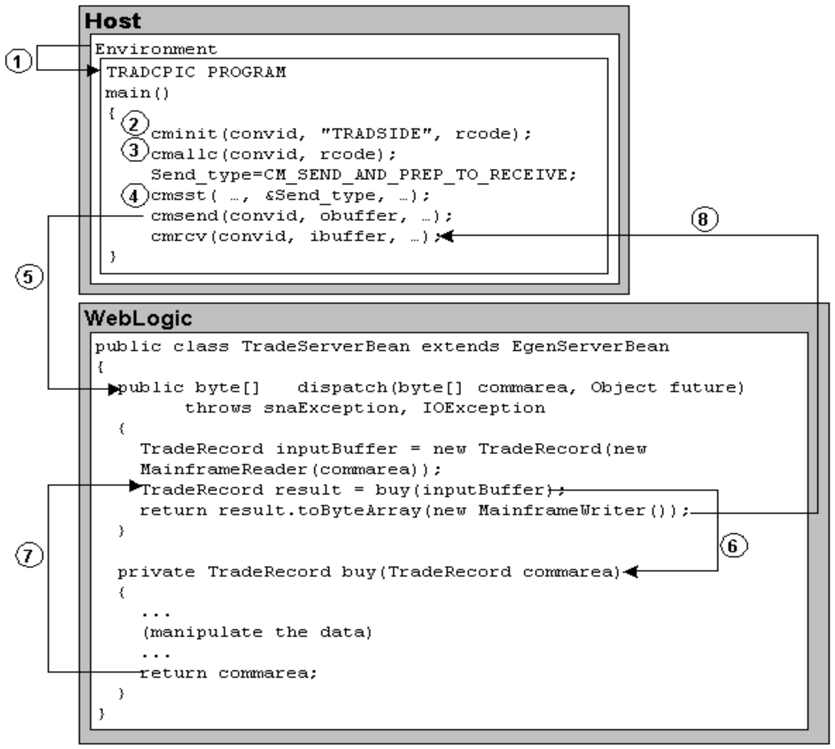
   Note: The BaseClient extends EgenClient, so the BaseClient inherits the callService method from EgenClient.

   The value of the first parameter is SIMPCPIC. SIMPCPIC is the name of the APPC Service that is mapped to the CPI-C Transaction Program ID TPNCPIC in the WebLogic Administration Console.
3. The transaction program TPNCPIC invokes the TOUPCPIC program.
4. TOUPCPIC accepts the conversation with the cmaccp call. The conversation ID returned in convid is used for all other requests on this conversation.
5. The cmrcv request receives the inputBuffer buffer contents for processing.
6. The TOUPCPIC program processes that data.

7. The cmsst request prepares for the send request by setting the send type to CM_SEND_AND_DEALLOCATE.
8. The cmsend request returns the obuffer contents. The data is returned from the EgenClient.callService method as the byte array rawResult.

Host CPI-C Request/Response to WebLogic Server EJB

The following figure shows a host CPI-C request/response to WebLogic Server EJB programming flow.



```
TRADCPIC PROGRAM
main()
{
    cminit(convid, "TRADSIDE", rcode);
    cmallc(convid, rcode);
    Send_type=CM_SEND_AND_PREP_TO_RECEIVE;
    cmsst( …, &Send_type, …);
    cmsend(convid, obuffer, …);
    cmrcv(convid, ibuffer, …);
}
```

```
public class TradeServerBean extends EgenServerBean
{
  public byte[]   dispatch(byte[] commarea, Object future)
        throws snaException, IOException
  {
    TradeRecord inputBuffer = new TradeRecord(new
    MainframeReader(commarea));
    TradeRecord result = buy(inputBuffer);
    return result.toByteArray(new MainframeWriter());
  }

  private TradeRecord buy(TradeRecord commarea)
  {
    ...
    (manipulate the data)
    ...
    return commarea;
  }
}
```

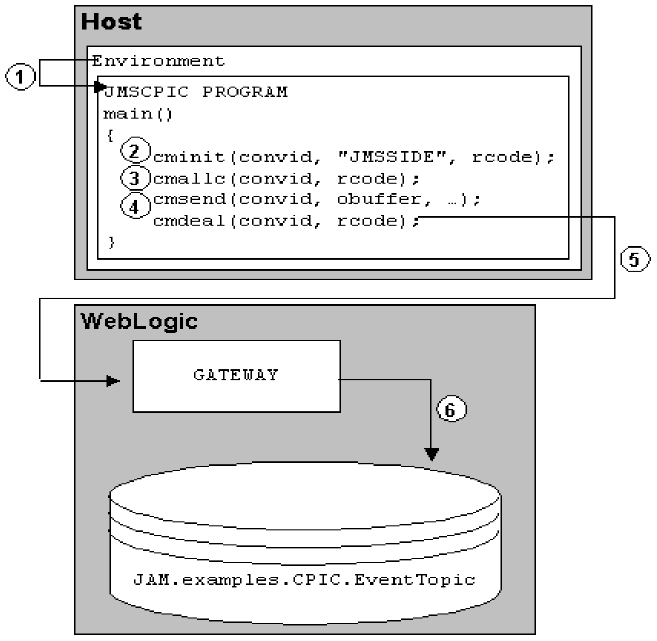
The following steps describe the host CPI-C request/response to WebLogic Server EJB programming flow.

1. The CPI-C application program TRADCPIC is invoked using the environment start-up specifications.
2. The TRADCPIC client requests cminit to establish conversation attributes and receive a conversation ID that will be used on all other requests on this conversation. The remote server and service are named in the CPI-C side information entry TRADSIDE.

3. The cmallc request initiates the advertised service TRADSERV. In the WebLogic Administration Console, the TRADSERV service is mapped to the JNDI name, jam.TradeServer, for the TradeServer EJB.
4. The cmsst request prepares the next send request by setting the send type to CM_SEND_AND_PREP_TO_RECEIVE.
5. The cmsend request immediately sends the contents of the obuffer to the dispatch method of TradeServerBean in the commarea byte array and relinquishes control.
6. The buy method is messaged from the dispatch method.
7. The business logic is performed, and the result is returned to the dispatch method.
8. The cmrcv request receives the contents of the byte array returned from the dispatch method in the ibuffer buffer, and notification that the conversation has ended with the return code value of CM_DEALLOCATED_NORMAL.

Host CPI-C Asynchronous No Reply to Java Server

The following figure shows a Host CPI-C asynchronous no reply to Java server programming flow.



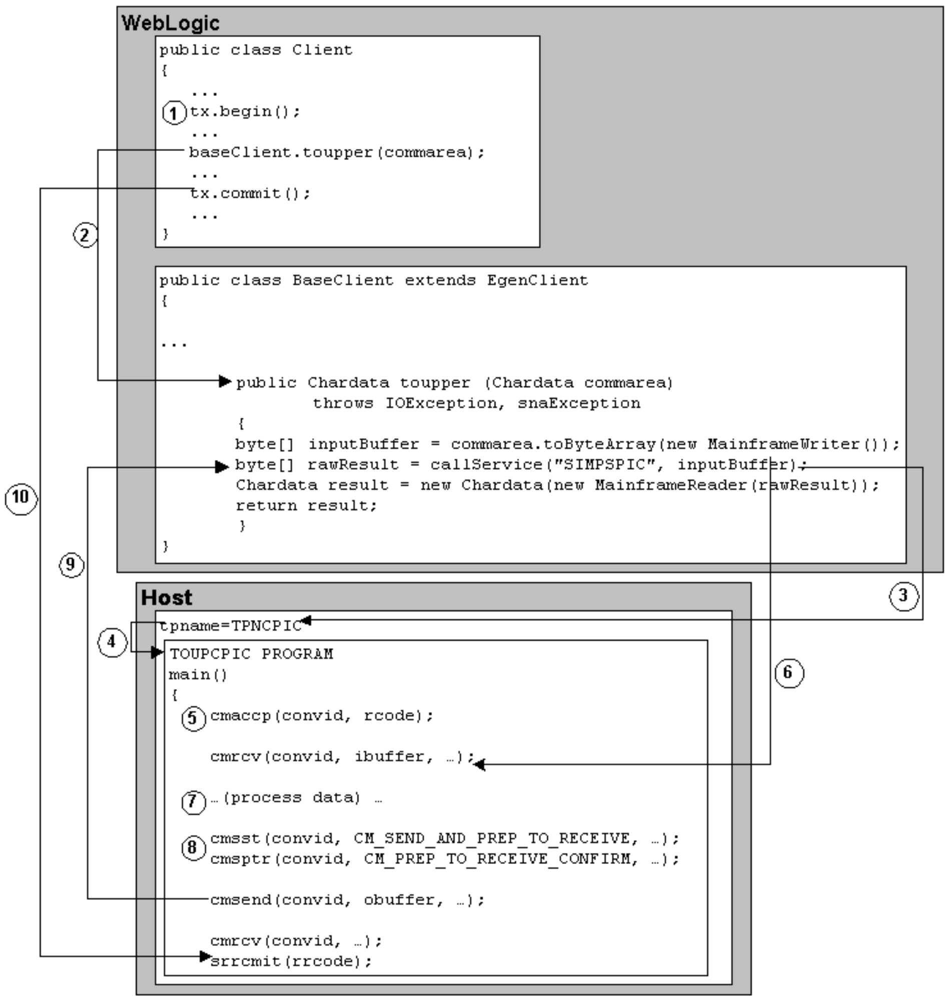
```
JMSCPIC PROGRAM
main()
{
    cminit(convid, "JMSSIDE", rcode);
    cmallc(convid, rcode);
    cmsend(convid, obuffer, …);
    cmdeal(convid, rcode);
}
```

The following steps describe the Host CPI-C asynchronous no reply to Java server programming flow.

1. The CPI-C application program JMSCPIC is invoked using the environment start-up specifications.
2. The JMSCPIC client requests cminit to establish conversation attributes and receive a conversation ID that will be used on all other requests on this conversation. The remote server and service are named in the CPI-C side information entry JMSSIDE.
3. The cmallc request initiates the advertised service CTOJMSSV.
4. The cmsend request sends the contents of the obuffer to the CTOJMSSV service.
5. The cmdeal request flushes the data and indicates the conversation is finished. The request message is forwarded to the Gateway.
6. The Gateway sends the message to the JMS Event CTOJMSSV. In the WebLogic Administration Console, the CTOJMSSV service name is mapped to the JMS topic, JAM.examples.CPIC.EventTopic.

Transactional Java Client Request/Response to Host CPI-C

The following figure shows a transactional Java client request/response to a Host CPI-C programming flow.



```
public class Client
{
   ...
   tx.begin();
   ...
   baseClient.toupper(commarea);
   ...
   tx.commit();
   ...
}
```

```
public class BaseClient extends EgenClient
{

...
         public Chardata toupper (Chardata commarea)
                  throws IOException, snaException
         {
         byte[] inputBuffer = commarea.toByteArray(new MainframeWriter());
         byte[] rawResult = callService("SIMPSPIC", inputBuffer);
         Chardata result = new Chardata(new MainframeReader(rawResult));
         return result;
         }
}
```

```
tpname=TPNCPIC
 TOUPCPIC PROGRAM
 main()
 {
    cmaccp(convid, rcode);

    cmrcv(convid, ibuffer, …);

    … (process data) …

    cmsst(convid, CM_SEND_AND_PREP_TO_RECEIVE, …);
    cmsptr(convid, CM_PREP_TO_RECEIVE_CONFIRM, …);

    cmsend(convid, obuffer, …);

    cmrcv(convid, …);
    srrcmit(rrcode);
```

The following steps describe the transactional Java client request/response to a host CPI-C programming flow.

1. A Java client class calls the begin method of a UserTransaction object to start a transaction.

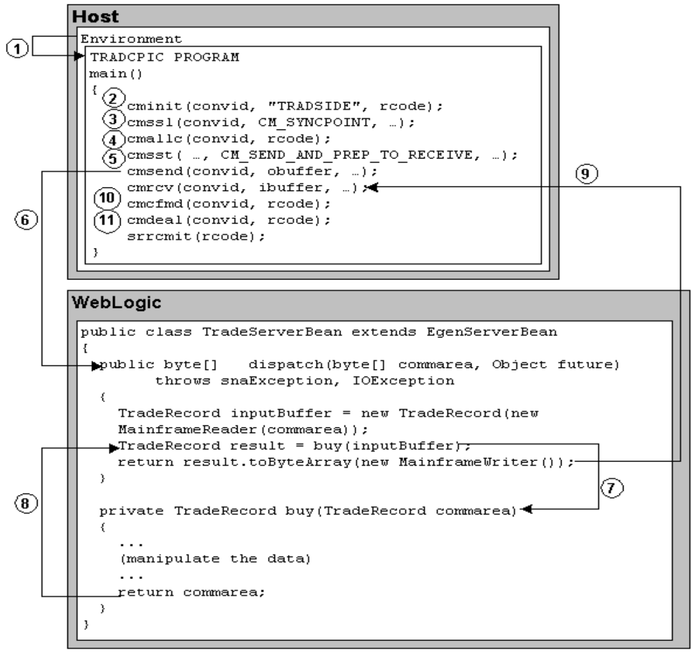
2. Within the boundaries of that transaction, the Java client class (stand-alone client, EJB, and so forth) makes a call to the BaseClient.toupper method with a Chardata DataView as the parameter.
3. In the toupper method, a call is made to the EgenClient.callService method.

   Note: The BaseClient extends EgenClient, so the BaseClient inherits the callService method from EgenClient.

   The value of the first parameter is SIMPCPIC. SIMPCPIC is the name of the APPC Service that is mapped to the CPI-C transaction program ID TPNCPIC in the WebLogic Administration Console.
4. The transaction program with the tpname TPNCPIC invokes the TOUPCPIC program.
5. TOUPCPIC accepts the conversation with the cmaccp call. The conversation ID returned in convid is used for all other requests on this conversation.
6. The cmrcv request receives the inputBuffer buffer contents for processing.
7. The TOUPCPIC program processes that data.
8. The cmsst and cmsptr prepare the next send request by setting the send type to CM_SEND_AND_PREP_TO_RECEIVE and by setting the prepare-to-receive type to CM_PREP_TO_RECEIVE_CONFIRM. The CONFIRM indicates that the service has completed successfully.
9. The cmsend request returns the obuffer contents. The data is returned from the EgenClient.callService method as the byte array rawResult.
10. The Java client class calls the commit method of the UserTransaction object to indicate the successful completion of the transaction and request the commit of all updated resources. The cmrcv request receives the commit request, and responds explicitly to the request with the SAA Resource/Recovery commit call srrcmit. The conversation is ended after the successful commit exchange.

Transactional Host CPI-C Request/Response to WebLogic Server EJB

The following figure shows a transactional host CPI-C request/response to WebLogic Server EJB programming flow.



```
TRADCPIC PROGRAM
main()
{
    cminit(convid, "TRADSIDE", rcode);
    cmssl(convid, CM_SYNCPOINT, …);
    cmallc(convid, rcode);
    cmsst( …, CM_SEND_AND_PREP_TO_RECEIVE, …);
    cmsend(convid, obuffer, …);
    cmrcv(convid, ibuffer, …);
    cmcfmd(convid, rcode);
    cmdeal(convid, rcode);
    srrcmit(rcode);
}
```

```
public class TradeServerBean extends EgenServerBean
{
  public byte[]   dispatch(byte[] commarea, Object future)
        throws snaException, IOException
  {
    TradeRecord inputBuffer = new TradeRecord(new
    MainframeReader(commarea));
    TradeRecord result = buy(inputBuffer);
    return result.toByteArray(new MainframeWriter());
  }

  private TradeRecord buy(TradeRecord commarea)
  {
    ...
    (manipulate the data)
    ...
    return commarea;
  }
}
```

The following steps describe the transactional host CPI-C request/response to WebLogic Server EJB programming flow.

1. The CPI-C application program TRADCPIC is invoked using the environment start-up specifications.
2. The TRADCPIC client requests cminit to establish conversation attributes and receive a conversation ID that will be used on all other requests on this conversation. The remote server and service are named in the CPI-C side information entry TRADSIDE.
3. The cmssl sets the conversation attribute to sync-level 2 with CM_SYNCPOINT. This allows the WebLogic EJB to participate in the transaction.
4. The cmallc request initiates the advertised service TRADSERV. In the WebLogic Administration Console the TRADSERV service is mapped to the JNDI name, jam.TradeServer for the TradeServer EJB.

5. The cmsst request prepares the next send request by setting the send type to CM_SEND_AND_PREP_TO_RECEIVE.
6. The cmsend request immediately sends the contents of the obuffer to the dispatch method of TradeServerBean in the commarea byte array and relinquishes control.
7. The buy method is messaged from the dispatch method.
8. The business logic is performed, and the result is returned to the dispatch method.
9. The cmrcv request receives the contents of the byte array returned from the dispatch method in the ibuffer buffer. The cmrcv receives a confirm request indicating the conversation should terminate.
10. The client replies positively to the confirm request with cmcfmd.
11. The TRADCPIC client prepares to free the conversation with the cmdeal request. The conversation in CM_DEALLOCATE_SYNC_LEVEL commits all updated resources in the transaction and waits for the SAA resource recovery verb, srrcmit. After the commit sequence has completed, the conversation terminates.

CHAPTER 6

Performing Your Own Data Translation

Topics:

This section discusses how to perform your own data translation in iWay JAM using the EgenClient.

Why Perform Your Own Data Translation?

The automatic data translation provided by DataViews can usually meet your requirements. The eGen-generated DataViews relieve your application of the burden of translating data between the mainframe EBCDIC environment and the Java runtime environment. In addition, native mainframe data types that are not supported in Java (such as packed, zoned decimal, etc.) are automatically mapped to appropriate Java data types. However, occasionally you may want to bypass these features and create your own data translation. The following are advantages of bypassing the eGen/DataView infrastructure:

- Unnecessary data translation may be avoided.

  If the data has been acquired in the appropriate format, it can simply be transmitted to the mainframe bypassing the DataView translation overhead.

- Contents of data buffer may be dynamically determined at run time.

  In some cases, this may be preferable to a DataView generated from a copybook containing numerous REDEFINES representing various record types.

Simple interfaces are provided for translating data both from and to the mainframe. In addition, a simple callService() method is available for making mainframe service requests.

Using EgenClient Directly

EgenClient is the iWay JAM class responsible for making service calls from WebLogic Server to the mainframe. This class is the foundation of all WebLogic Server to Mainframe communication by eGen-created EJB and Servlet objects. EgenClient may also be used directly by applications to issue mainframe service requests. The following EgenClient Public Interface shows the public methods available for your use:[Stefan: Please check wording of this sentence]

```
package com.iwaysoftware.jam.egen;

import java.io.IOException;
import com.iwaysoftware.jam.sna.jcrmgw.snaException;

public class EgenClient
{
public EgenClient();
public EgenClient(String serverURL);
public void setServerURL(String serverURL);
public byte[] callService(String service, byte[] in)
throws snaException, IOException;
public void setUserID(String userid);
public void setPassword(String password);
}
```

The following tables lists and describes these methods.

Method	Description
EgenClient()	The default constructor. Constructing an EgenClient class using the default constructor will search for a jam.url property containing the iWay JAM Gateway server URL.
EgenClient(URL)	If the EgenClient class is provided a URL at construction, it is used in place of the search for a jam.url property.
setServerURL(URL)	May be used to override the URL set at construction. All service calls following the invocation of this method will use the URL provided.
callService(service, in)	The workhorse of the EgenClient class. The mainframe service in the iWay JAM configuration named SERVICE will be called and passed the buffer provided by the IN parameter. The response buffer of the service is returned from this method.
setUserID(userid)	Sets the User ID used to access a mainframe service.
setPassword(password)	Sets the password used to access a mainframe service.

How EgenClient Locates an iWay JAM Gateway

The EgenClient class requires a connection to a WebLogic Server running an iWay JAM Gateway to communicate with a mainframe. This connection is accomplished via a URL provided by the caller identifying the server, or cluster of servers, hosting the iWay JAM Gateway(s). The EgenClient class attempts to obtain this URL from the following sources (listed in priority order):

1. If the EgenClient.setServerURL() method has been called, the URL provided is used to locate an iWay JAM Gateway.
2. If a URL was provided on the EgenClient constructor, this URL is used to locate a iWay JAM Gateway.
3. EgenClient checks for the existence of a jam.url system property and, if present, uses its value as the URL to locate an iWay JAM gateway.
4. EgenClient searches the CLASSPATH for a file named jam.properties. If this properties file is found and contains a jam.url entry, this value is used to locate an iWay JAM Gateway.
5. EgenClient assumes that it is running on the same WebLogic Server as the iWay JAM Gateway and attempts to establish a local connection.

Using EgenClient to Make a Mainframe Request

The following illustrates how to call a mainframe service using the EgenClient class. This example assumes that a properly formatted mainframe buffer is passed as a parameter, and that the URL of a correctly configured iWay JAM Gateway is set via the jam.url property.

```
import com.iwaysoftware.jam.egen.EgenClient;
import com.iwaysoftware.jam.sna.jcrmgw.snaException;
import java.io.IOException;
.
.
public byte[] getPurchaseOrder(byte[] poNum)
throws IOException
{
try
{
return(new EgenClient().callService("GetPO", poNum));
}
catch (snaException e)
{
throw new IOException(e.getMessage());
}
}
```

The sections that follow provide information on dynamically creating mainframe buffers and interpreting the responses from mainframe services.

Translating Buffers from Java to Mainframe Representation

Support for creating buffers for input to a mainframe service is provided by the com.iwaysoftware.jam.base.io.MainframeWriter class. This class functions similar to a Java java.io.DataOutputStream object. It translates Java data types and all mainframe-native data types. For numeric data types, this translation service provides a conversion from Java native numeric types to those available on the mainframe. For string data types, a translation is performed from UNICODE to EBCDIC by default, although the output codepage used is configurable.

MainframeWriter Public Interface

The following shows the public methods provided by the MainframeWriter class.

```
package com.iwaysoftware.jam.base.io;

public class MainframeWriter
{
public MainframeWriter();
public MainframeWriter(String codepage);
public void setDefaultCodepage(String cp)
public byte[] toByteArray();
public void writeRaw(byte[] bytes
throws IOException;
public void writeFloat(float value)
throws IOException;
public void writeDouble(double value)
throws IOException;
public void write(char c)
throws IOException;
public void writePadded(String s, char padChar, int length)
throws IOException;
public void write16bit(int value)
throws IOException;
public void write16bitUnsigned(int value)
throws IOException;
public void write16bit(long value, int scale)
throws IOException, ArithmeticException;
public void write16bitUnsigned(long value, int scale)
throws IOException, ArithmeticException;
public void write32bit(int value)
throws IOException;
public void write32bitUnsigned(long value)
throws IOException;
public void write32bit(long value, int scale)
throws IOException, ArithmeticException;
public void write32bitUnsigned(long value, int scale)
throws IOException, ArithmeticException;
public void write64bit(long value)
throws IOException;
public void write64bitUnsigned(long value)
throws IOException;
public void write64bitBigUnsigned(BigDecimal value)
throws IOException;
public void write64bit(long value, int scale)
throws IOException, ArithmeticException;
public void write64bit(BigDecimal value, int scale)
throws IOException, ArithmeticException;
public void write64bitUnsigned(long value, int scale)
throws IOException, ArithmeticException;
public void write64bitUnsigned(BigDecimal value, int scale)
throws IOException, ArithmeticException;
public void writePacked(BigDecimal value, int digits,
int precision, int scale)
throws ArithmeticException, IOException;
public void writePackedUnsigned(BigDecimal value,
int digits, int precision, int scale)
throws ArithmeticException, IOException;
}
```

The following table lists and describes these methods.

Method	**Description**
MainframeWriter()	The default constructor. Constructs a MainframeWriter using the default code page of cp037 (EBCDIC).
MainframeWriter(cp)	Constructs a MainframeWriter using the specified codepage for character field translation.
setDefaultCodepage(cp)	Set the codepage to be used for all future data translations.
toByteArray()	Returns the translated buffer constructed by writing data to the MainframeWriter class as a byte array.
writeRaw(bytes)	Write a raw byte array to the output buffer.
writeFloat(num)	Convert a floating point number from the IEEE Java float data type to IBM 4 byte floating point format. The equivalent COBOL picture clause is PIC S9V9 COMP-1.
writeDouble(num)	Convert a floating point number from the IEEE Java double data type to IBM 8 byte floating point format. The equivalent COBOL picture clause is PIC S9V9 COMP-2.
write(c)	Translate and write a single character to the output buffer. The equivalent COBOL picture clause is PIC X.

Method	**Description**
writePadded(str, pad, len)	Translate and write a string to a fixed length character field. The passed pad character is used if the length of the passed string is less than len. If greater than len, it is truncated to len characters. The equivalent COBOL picture clause is PIC X(len).
write16bit(num)	Writes a signed 16 bit binary integer to the output buffer. The equivalent COBOL picture clause is PIC S9(4) COMP.
write16bitUnsigned(num)	Writes an unsigned 16 bit binary integer to the output buffer. The equivalent COBOL picture clause is PIC 9(4) COMP.
write16bit(num, scale)	Writes a signed 16 bit integer to the output buffer after moving the implied decimal point left by scale digits. For example, the call write16bit(100, 1) results in the value 10 being written. The equivalent COBOL picture clause is PIC S9(4) COMP.
write16bitUnsigned(num, scale)	Writes an unsigned 16 bit integer to the output buffer after moving the implied decimal point left by `scale` digits. For example, the call write16bitUnsigned(100, 1) results in the value 10 being written. The equivalent COBOL picture clause is PIC 9(4) COMP.
write32bit(num)	Writes a signed 32 bit binary integer to the output buffer. The equivalent COBOL picture clause is PIC S9(8) COMP.
write32bitUnsigned(num)	Writes an unsigned 32 bit binary integer to the output buffer. The equivalent COBOL picture clause is PIC 9(8) COMP.
write32bit(num, scale)	Writes a signed 32 bit integer to the output buffer after moving the implied decimal point left by scale digits. For example, the call write32bit(100L, 1) results in the value 10 being written. The equivalent COBOL picture clause is PIC S9(8) COMP.

Method	**Description**
write32bitUnsigned(num, scale)	Writes an unsigned 32 bit integer to the output buffer after moving the implied decimal point left by scale digits. For example, the call write32bitUnsigned(100L, 1) results in the value 10 being written. The equivalent COBOL picture clause is PIC 9(8) COMP.
write64bit(num)	Writes a signed 64 bit binary integer to the output buffer. The equivalent COBOL picture clause is PIC S9(15) COMP.
write64bitUnsigned(num)	Writes an unsigned 64 bit binary integer to the output buffer. The equivalent COBOL picture clause is PIC 9(15) COMP.
write64bit(num, scale)	Writes a signed 64 bit integer to the output buffer after moving the implied decimal point left by scale digits. For example, the call write64bit(100L, 1) results in the value 10 being written. The equivalent COBOL picture clause is PIC S9(15) COMP.
write64bitUnsigned(num, scale)	Writes an unsigned 64 bit integer to the output buffer after moving the implied decimal point left by scale digits. For example, the call write64bitUnsigned(100L, 1) results in the value 10 being written. The equivalent COBOL picture clause is PIC 9(15) COMP.
writePacked(num, digits, prec, scale)	Writes a decimal number as an IBM signed packed data type with digits decimal digits total and prec digits to the right of the decimal point. Prior to conversion, the number is scaled to the left scale digits. The equivalent COBOL picture clause is PIC S9(digits-prec)V9(prec) COMP-3.
writePackedUnsigned(num, digits, prec, scale)	Writes a decimal number as an IBM unsigned packed data type with digits decimal digits total and prec digits to the right of the decimal point. Prior to conversion the number is scaled to the left scale digits. The equivalent COBOL picture clause is PIC 9(digits-prec)V9(prec) COMP-3.

Using MainframeWriter to Create Data Buffers

As an example of using the MainframeWriter class to create a mainframe data buffer, assume we have a mainframe service which accepts the following data record.

```
01 INPUT-DATA-REC.
       05  FIRST-NAME       PIC X(10).
       05  LAST-NAME        PIC X(10).
       05  AGE              PIC S9(4) COMP.
       05  HOURLY-RATE      PIC S9(3)V9(2) COMP-3.
```

The following shows a Java test program that creates a buffer matching this record layout using the MainframeWriter translation class:

```
import java.math.BigDecimal;

import com.iwaysoftware.jam.base.io.MainframeWriter;

public class MakeBuffer
{
public static void main(String[] args) throws Exception
{
MainframeWriter mf = new MainframeWriter();
mf.writePadded("Edgar", ' ', 10);// first name
mf.writePadded("Jones", ' ', 10);// last name
mf.write16bit(22);// age
mf.writePacked(new BigDecimal(22.50), 5, 2, 0);// hourly rate
byte[] buffer = mf.toByteArray();
System.out.println(getHexString(buffer));
}

private static String getHexString(byte[] buffer)
{
StringBuffer hexStr = new StringBuffer(buffer.length * 2);
for (int i = 0; i < buffer.length; ++i)
{
int n = buffer[i] & 0xff;
hexStr.append(hex[n >> 4]);
hexStr.append(hex[n & 0x0f]);
}
return(hexStr.toString());
}

private static char[] hex =
{
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
};
}
```

The output of running this sample program is:

```
C5848781994040404040D1969585A24040404040001602250C
```

This buffer breaks down as follows:

```
FIRST-NAMEC5848781994040404040"Edgar" + 5 spaces in EBCDIC
LAST-NAMED1969585A24040404040"Jones" + 5 spaces in EBCDIC
AGE001622 as 16 bit integer
HOURLY-RATE02250C22.50 positive packed number
(decimal point is assumed)
```

Translating Buffers from Mainframe Format to Java

Support for translating data received from the mainframe to Java data types is provided by the com.iwaysoftware.jam.base.io.MainframeReader class. This class operates in a manner similar to a Java jam.io.DataInputStream, and performs translations from mainframe data types to equivalent types usable by a Java program. Like the MainframeWriter class, the codepage used for string translations may be configured and defaults to EBCDIC.

MainframeReader Public Interface

The following shows the public methods provided by the MainframeReader class.

```
package com.iwaysoftware.jam.base.io;

public class MainframeReader
{
public MainframeReader(byte[] buffer);
public MainframeReader(byte[] buffer, String codepage);
public void setDefaultCodepage(String cp);
public byte[] readRaw(int count) throws IOException;
public float readFloat() throws IOException;
public double readDouble() throws IOException;
public char readChar() throws IOException;
public String readPadded(char padChar, int length)
throws IOException;
public short read16bit() throws IOException;
public int read16bitUnsigned() throws IOException;
public long read16bit(int scale) throws IOException;
public int read32bit() throws IOException;
public long read32bit(int scale)
throws IOException;
public long read32bitUnsigned() throws IOException;
public long read32bitUnsigned(int scale) throws IOException;
public long read64bit() throws IOException;
public long read64bitUnsigned()
throws IOException;
public long read64bit(int scale)
throws IOException;
public BigDecimal read64bitBigUnsigned()
throws IOException;
public BigDecimal read64bitBig(int scale)
throws IOException
public BigDecimal readPackedUnsigned(int digits,
int precision, int scale)
throws ArithmeticException, IOException;
public BigDecimal readPacked(int digits,
int precision, int scale)
throws ArithmeticException, IOException;
}
```

The following table lists and describes these methods.

Method	**Description**
MainframeReader(buffer)	Constructs a MainframeReader for the passed buffer using the default code page of `cp037` (EBCDIC).

Method	**Description**
MainframeReader(buffer, cp)	Constructs a MainframeReader for the passed buffer using the specified codepage for character field translation.
setDefaultCodepage(cp)	Sets the codepage to be used for all future character translations.
readRaw(count)	Read count characters from the buffer without any translation and return them as a byte array.
readFloat()	Read a 4 byte IBM floating point number and return it as a Java float data type.
readDouble()	Read an 8 byte IBM floating point number and return it as a Java double data type.
readChar()	Read and translate a single character.
readPadded(pad, len)	Read and translate a fixed length character field and return it as a Java String. The length of the field is passed as len and the field pad character is passed as pad. Trailing instances of the pad character are removed before the data is returned.
read16bit()	Read a 16 bit binary integer and return it as a Java short.
read16bitUnsigned()	Read an unsigned 16 bit integer and return it as a Java int.
read16bit(scale)	Read a 16 bit binary integer and scale the value by 10^scale. For example, if the value 10 is read via read16bit(1), the returned value would be 100.
read32bit()	Read a 32 bit binary integer and return it as a Java int.
read32bit(scale)	Read a 32 bit binary integer and scale the value by 10^scale. For example, if the value 10 is read via read32bit(1), the returned value would be 100.
read32bitUnsigned()	Read an unsigned 32 bit integer and return it as a Java long.
read32bitUnsigned(scale)	Read an unsigned 32 bit binary integer and scale the value by 10^scale. For example, if the value 10 is read via read32bit(1), the returned value would be 100.

Method	**Description**
read64bit()	Read a 64 bit binary integer and return it as a Java long.
read64bitUnsigned()	Read an unsigned 64 bit integer and return it as a Java long.
read64bitUnsigned(scale)	Read an unsigned 64 bit binary integer and scale the value by 10^scale. For example, if the value 10 is read via read32bit(1), the returned value would be 100.
read64bitBigUnsigned()	Read an unsigned 64 bit integer and return it as a Java BigDecimal.
read64bitBig(scale)	Read a signed 64 bit integer and scale the value by 10^scale. The value is returned as a Java BigDecimal.
readPackedUnsigned(digits, prec, scale)	Read an unsigned packed number consisting of `digits` numeric digits with `prec` digits to the right of the decimal. The value is scaled by 10^scale returned as a Java BigDecimal.
readPacked(digits, prec, scale)	Read a signed packed number consisting of digits numeric digits with prec digits to the right of the decimal. The value is scaled by 10^scale returned as a Java BigDecimal.

Using MainframeReader to Translate Data Buffers

As an example of using the MainframeReader, class following is a program that translates and displays the fields in the mainframe buffer created previously. Our input buffer consists of the binary data:

```
C58487819940404040D1969585A240404040400001602250C
```

The following shows the sample program used to process this buffer:

```
import java.math.BigDecimal;
import com.iwaysoftware.jam.base.io.MainframeReader;

public class ShowBuffer
```

```
{
public static void main(String[] args) throws Exception
{
String data =
"C5848781994040404040D1969585A24040404040001602250C";
byte[] buffer = buildBinary(data);
MainframeReader mf = new MainframeReader(buffer);
System.out.println(" First Name: " + mf.readPadded(' ', 10));
System.out.println("  Last Name: " + mf.readPadded(' ', 10));
System.out.println("        Age: " + mf.read16bit());
System.out.println("Hourly Rate: " + mf.readPacked(5, 2, 0));
}

private static byte[] buildBinary(String data)
{
byte[] buffer = new byte[data.length() / 2];
for (int i = 0; i < buffer.length; ++i)
{
int msb = hex.indexOf(data.charAt(i * 2));
int lsb = hex.indexOf(data.charAt(i * 2 + 1));
buffer[i] = (byte) (msb << 4 | lsb);
}
return(buffer);
}

private static final String hex = "0123456789ABCDEF";
}
```

When run, the program produces the following output:

```
First Name: Edgar
Last Name: Jones
Age: 22
Hourly Rate: 22.50
```

CHAPTER 7

Diagnostics

Topics:

- Gateway Statistics
- Gateway Tracing
- CRM Tracing
- APPC API Tracing

This section discusses diagnostics and tracing.

Gateway Statistics

You can display the statistics for a Gateway definition using the JAM Administration Console. For instructions on accessing Gateway statistics, refer to the *iWay Java Adapter for Mainframe Configuration and Administration Guide*.

The following table lists the statistics information for the Gateway.

Total Requests	Number of requests that reached the gateway. May be larger than the sum of successes and failures if some requests are still being processed.
Total Successes	Number of requests that were successfully processed to completion by the gateway. Application level failures may be reported as gateway successes.
Average Response Time	Average response time for all successful requests and some failures. Failures that fail before they are transmitted over the network do not affect this statistic. Timeouts do not affect this statistic until a late reply is received.
Total Failures	Total number of failures of any kind.
No Response	Number of requests that timed out and never received a response.
Late Response	Number of requests that timed out and then received a response.
Other	Number of request that failed other than by timeout.

Gateway Tracing

iWay JAM run-time traces are sent to the WebLogic log as “Debug” messages. Debug messages are written to each WebLogic Server log file but are not sent to the administration server. In addition, these messages are only sent to the server stdout if the server configuration has both the Log to Stdout and Debug to Stdout options selected on the server Logging/General page.

For instructions on accessing Gateway tracing options, see the *iWay Java Adapter for Mainframe Configuration and Administration Guide*.

The following table lists the user trace categories for the Gateway.

User level trace	Produces trace records for the beginning and completion of all user requests, both to and from the mainframe. The completion message will indicate the success or failure of the request.

User dump trace	Produces trace records with a hexadecimal dump of the user data associated with all user requests and replies. This trace level will also cause the trace records for User level trace to be produced.

The following is an example of a trace for two user requests:

```
<Nov 15, 2001 3:53:06 PM GMT-06:00> <Debug> <JAM1> <[5560199] Beginning of
request:134217866 service:sampleCreate>

<Nov 15, 2001 3:53:06 PM GMT-06:00> <Debug> <JAM1> <[5560199] ---- request data dump
----

0000: 00 00 00 00 0f d3 81 a2 a3 61 f0 40 40 40 40 40   .....Last/0
0010: 40 40 40 40 c6 89 99 a2 a3 61 f1 40 40 40 40 40       First/1
0020: 40 40 40 d4 f3 f2 f0 f0 40 c1 95 a8 a2 a3 99 85      M3200 Anystre
0030: 85 a3 40 c3 96 a4 99 a3 40 40 40 40 40 40 40 40   et Court
0040: 40 40 e3 e7 f7 f7 f5 f5 f5 f0 f0 f0 f0              TX775550000
------------------------------
>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] End of
request:134217866>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] ---- response data
dump ----

0000: 00 00 00 00 0f d3 81 a2 a3 61 f0 40 40 40 40 40   .....Last/0
0010: 40 40 40 40 c6 89 99 a2 a3 61 f1 40 40 40 40 40       First/1
0020: 40 40 40 d4 f3 f2 f0 f0 40 c1 95 a8 a2 a3 99 85      M3200 Anystre
0030: 85 a3 40 c3 96 a4 99 a3 40 40 40 40 40 40 40 40   et Court
0040: 40 40 e3 e7 f7 f7 f5 f5 f5 f0 f0 f0 f0              TX775550000
------------------------------
>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] Starting one phase
commit>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] Beginning of
request:1207959692 service:sampleRead>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] ---- request data dump
----

0000: 00 00 00 00 0f d3 81 a2 a3 61 f0 40 40 40 40 40   .....Last/0
0010: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
0020: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
0030: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
0040: 40 40 40 40 40 40 40 40 40 40 40 40 40
------------------------------
>
```

```
<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] End of
request:1207959692>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] ---- response data
dump ----

0000: 00 00 00 00 0f d3 81 a2 a3 61 f0 40 40 40 40 40   .....Last/0
0010: 40 40 40 40 c6 89 99 a2 a3 61 f1 40 40 40 40 40      First/1
0020: 40 40 40 d4 f3 f2 f0 f0 40 c1 95 a8 a2 a3 99 85     M3200 Anystre
0030: 85 a3 40 c3 96 a4 99 a3 40 40 40 40 40 40 40 40   et Court
0040: 40 40 e3 e7 f7 f7 f5 f5 f5 f0 f0 f0 f0              TX775550000
------------------------------
>

<Nov 15, 2001 3:53:07 PM GMT-06:00> <Debug> <JAM1> <[5560199] Starting one phase
commit>
```

The trace categories are for use if you find it necessary to contact BEA Technical Support. They may be used to collect data about your system necessary to resolve problems.

The following table lists the system trace categoies.

CRMAPI trace	Produces trace records showing the messages exchanged between the Gateway and the CRM.
JAM socket trace	Produces trace records showing a hexadecimal dump of the data exchanged between the Gateway and the CRM.
Configuration trace	Produces trace records showing operations within the JAM Administration Console and interactions between it and the Gateway.
Thread level trace	Produces trace records showing operations within the Gateway related to its internal threads and subtasks.

CRM Tracing

The CRM has tracing options that can be enabled for advanced debugging of iWay JAM applications. Refer to the *iWay Java Adapter for Mainframe Configuration and Administration Guide* for information about setting trace levels.

On Windows NT and Unix systems, traces are written to a file in the directory in which the CRM was started. If the environment variable `APPDIR` is set, the trace will be written to the directory it specifies. The file name will be specified as:

```
CRM.<pid>.trace.<seq>
```

Where *<pid>* is the process ID of the CRM process, and *<seq>* is the sequence number of the trace file, which is always 0.

On MVS systems, traces are written to SYSOUT, which is identified by TRACE DD NAME.

Viewing Trace Output

With a few exceptions, each line in the trace output is preceded by a time tag, identifying the date and time the line was written.

Note: The time tag information in the CRM trace should reflect the current system time. In order to make use of the correct time zone information on Unix and MVS systems, it is important that the TZ environment variable be set correctly. If this variable is not set correctly on your system, refer to your system documentation for further information.

After the time tag, a four-digit number appears, identifying the number of the task that wrote the line to the trace. This number can be useful when multiple processes are connected to the CRM.

If the trace level of the CRM is greater than one, a plus sign (+) following the task number indicates that a line in the trace is level 1 output. For example, in the sequence:

```
Tue Oct 09 10:45:10.291 0001 +CRM initialization complete -- Normal
dispatching begins

Tue Oct 09 10:45:10.291 0001  CRM state transition from
InitializationInProgress to Reset
```

The line `CRM initialization complete` is level 1 output, and the line `CRM state transition` is not (it is level 3 output).

When the trace level is set to 3, hex dump information will appear in the trace. These entries will appear interspersed with other trace statements, as shown in the following example.

```
OFFS  ----------------- HEXADECIMAL------------------  *-------ASCII-----*
0000: 00 00 00 B2 63 00 00 56 BE AC 05 00 00 04 00 02  (....c..V........)
0010: 00 00 00 00 00 00 00 1C 7E 71 00 00 00 00 00 96  (.........q......)
0020: 7E 76 00 00 41 30 36 52 65 67 69 6F 6E 00 00 00  (.v..A06Region...)
0030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  (................)
0040: 00 00 00 00 01 57 45 42 4C 00 43 49 43 53 00 53  (.....WEBL.CICS.S)
0050: 4E 41 43 52 4D 00 00 00 00 00 00 00 00 00 00 00  (NACRM...........)
0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 41 30  (..............A0)
0070: 36 43 49 43 53 00 00 00 00 00 00 00 00 00 00 00  (6CICS...........)
0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 41 30 36  (.............A06)
0090: 43 49 43 53 00 00 53 4D 53 4E 41 31 30 30 00 4C  (CICS..SMSNA100.L)
00A0: 4F 43 41 4C 00 00 00 00 00 00 02 00 00 04 00 02  (OCAL............)
00B0: EA 60                                            (..)
```

These entries consist of offset information in the left column, followed by columns with the data in hexadecimal format, followed by an ASCII or EBCDIC representation of the data. The data is read from left to right, top to bottom.

Hex dump information for application data appears in a slightly different format, with two different representations of the user data. An example follows:

```
00000 |12345678 9fe29489 a3884040 40404040| |.....Smith        |
00010 |40404040 d1968895 40404040 40404040| |    John          |
00020 |404040d8 f1f2f3f4 40c59394 40e2a34b| |   Q1234 Elm St.|
00030 |40404040 40404040 40404040 40404040| |                  |
00040 |4040e3d5 f1f2f3f4 f5404040 40000000| |  TN12345     ...|
```

The two columns following the hex data contain the user data in "actual" and "native" representations. In the "actual" representation, the binary data is represented directly as character data, with unprintable characters appearing as a period (.). In the "native" representation, the binary data is converted to the native character format (EBCDIC or ASCII), allowing text fields to be viewed directly.

Note: The above example was taken from a CRM trace from an EBCDIC machine, so the "actual" and "native" columns both contain readable text.

APPC API Tracing

The VTAM APPC API may be captured by enabling the APPC API tracing. The API trace shows the parameters and values passed and returned to the VTAM APPC stack. The API trace is captured to the GTF tracing facility. The GTF tracing facility must be active in the mainframe region to capture the API traces.

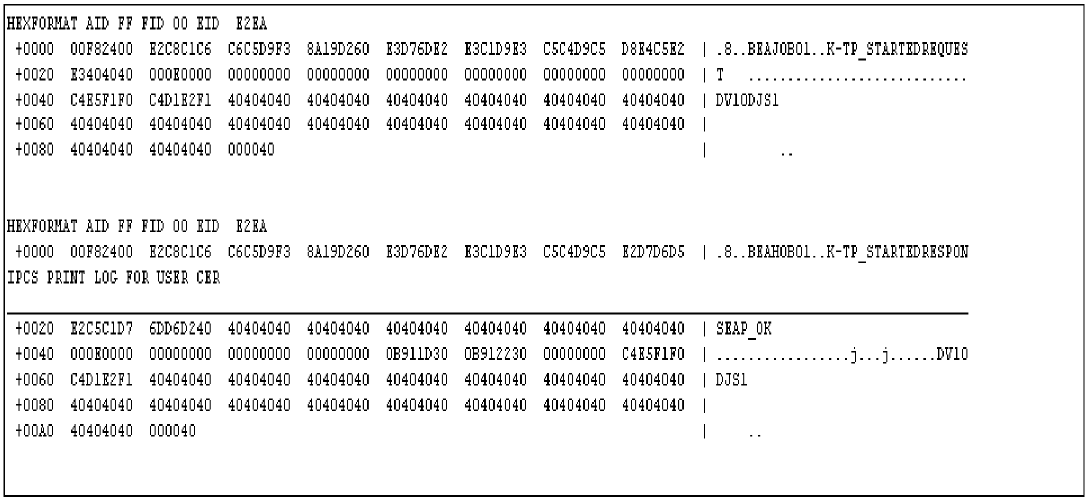
After capturing the traces, you must format the print using GTF formatting procedures such as IPCS. The APPC API trace is written to GTF as user id '2EA'. You may use this ID to filter the GTF print to include only the APPC API traces.

Refer to the *iWay Java Adapter for Mainframe Configuration and Administration Guide* for information about setting APPC tracing.

Viewing APPC Trace Output

The APPC API trace captures the parameters and values used by the CRM to make a VTAM APPC request. The trace will show the APPC verb control block before and after the request is made. The response to the request will show return codes and returned values.

The following example of a request and a response was formatted by using the IBM provided program IKJEFT01.



```
HEXFORMAT AID FF FID 00 EID  E2EA
 +0000  00F82400  E2C8C1C6  C6C5D9F3  8A19D260  E3D76DE2  E3C1D9E3  C5C4D9C5  D8E4C5E2  | .8..BEAJ0B01..K-TP_STARTEDREQUES
 +0020  E3404040  000E0000  00000000  00000000  00000000  00000000  00000000  00000000  | T   ...........................
 +0040  C4E5F1F0  C4D1E2F1  40404040  40404040  40404040  40404040  40404040  40404040  | DV10DJS1
 +0060  40404040  40404040  40404040  40404040  40404040  40404040  40404040  40404040  |
 +0080  40404040  40404040  000040                                                      |         ..


HEXFORMAT AID FF FID 00 EID  E2EA
 +0000  00F82400  E2C8C1C6  C6C5D9F3  8A19D260  E3D76DE2  E3C1D9E3  C5C4D9C5  E2D7D6D5  | .8..BEAH0B01..K-TP_STARTEDRESPON
IPCS PRINT LOG FOR USER CER
____________________________________________________________________________________________________________________
 +0020  E2C5C1D7  6DD6D240  40404040  40404040  40404040  40404040  40404040  40404040  | SEAP_OK
 +0040  000E0000  00000000  00000000  00000000  0B911D30  0B912230  00000000  C4E5F1F0  | ................j...j......DV10
 +0060  C4D1E2F1  40404040  40404040  40404040  40404040  40404040  40404040  40404040  | DJS1
 +0080  40404040  40404040  40404040  40404040  40404040  40404040  40404040  40404040  |
 +00A0  40404040  000040                                                                |     ..
```

APPENDIX A

DataView Programming Reference

Topics:

- Field Name Mapping Rules
- Field Type Mappings
- Elementary Field Accessors
- Array Field Accessors
- Fields with REDEFINES Clauses
- COBOL Data Types
- Other Access Methods for Generated DataView Classes
- Known Limitations of iWay JAM working with COBOL Copybooks

This section provides the rules that allow you to identify what form a generated Java class takes from a given COBOL copybook processed by the eGen Application Generator (eGen utility). An understanding of the rules facilitates a programmer's ability to correctly code any custom programs that make use of the generated classes.

The eGen utility maps a COBOL copybook into a Java class. The COBOL copybook contains a data record description. The eGen utility derives the generated Java class from the com.iwaysoftware.jam.dmd.dataview.DataView class (later referred to as DataView), which is provided on your iWay JAM product CD-ROM in the iwjam.jar file.

You should find the COBOL terms in this section easy to understand; however, you may need to use a COBOL reference book or discuss the terms with a COBOL programmer. Also, you can process a copybook with the eGen utility and examine the generated Java code in order to understand the mapping.

Field Name Mapping Rules

When you process a COBOL copybook containing field names, they are mapped to Java names by the eGen utility. All alphabetic characters are mapped to lower case, except in the following two cases.

- All dashes are removed and the character following the dash is mapped to upper case.
- When a prefix is added to the name (as when creating a field accessor function name), the first character of the base name is mapped to upper case.

The following table lists mapping examples.

COBOL Field Name	Java Base Name	Sample Accessor Name
EMP-REC	empRec	setEmpRec
500-REC-CNT	500RecCnt	set500RecCnt

Field Type Mappings

When you process a COBOL copybook, the data types of fields are mapped to Java data types. The mapping is performed by the eGen utility according to the following rules:

1. Groups map to DataView subclasses.
2. All alphanumeric fields are mapped to type String.
3. All edited numeric fields are mapped to type String.
4. All SIGN SEPARATE, BLANK WHEN ZERO or JUSTIFIED RIGHT fields are mapped to type String.
5. SIGN IS LEADING is not supported.
6. The types COMP-1, COMP-2, COMP-5, COMP-X, and PROCEDURE-POINTER fields are not supported (an error message is generated).
7. All INDEX fields are mapped to Java type INT.
8. POINTER maps to Java type INT.
9. All numeric fields with any digits to the right of the decimal point are mapped to type BigDecimal.
10. All COMP-3 (packed) fields are mapped to type BigDecimal.

11. All other numeric fields are mapped as shown in the following table.

Number of Digits	Java Type
<= 4	short
> 4 and <= 9	int
> 9 and <= 18	long
> 18	BigDecimal

Group Field Accessors

Each nested group in a COBOL copybook is mapped to a corresponding DataView subclass. The generated subclasses are nested exactly as the COBOL groups in the copybook. In addition, the eGen utility generates a private instance variable of this class type and a get accessor.

For example, the following copybook:

```
10 MY-RECORD.
20  MY-GRP.
30  ALNUM-FIELDPIC X(20).
```

```
Produces code similar to the following:
```

```
public MyGrp2V getMyGrp();
public static class MyGrp2V extends DataView
{
// Class definition
}
```

Elementary Field Accessors

Each elementary field is mapped to a private instance variable within the generated DataView subclass. Access to this variable is accomplished by two accessors that are generated (SET and GET).

These accessors have the following forms:

```
public void setFieldName(FieldType value);
```

```
public FieldType getFieldName();
```

where:

`FieldType`

Is described in *Field Type Mappings* on page A-2.

`FieldName`

Is described in *Field Type Mappings* on page A-2.

For example, the following copybook:

```
10 MY-RECORD.
20  NUMERIC-FIELDPIC S9(5).
20  ALNUM-FIELDPIC X(20).
```

produces the accessors:

```
public void setNumericField(int value);
public int getNumericField();
public void setAlnumField(String value);
public String getAlnumField();
```

Array Field Accessors

Array fields are handled according to the field accessor rules described in *Group Field Accessors* on page A-3 and *Elementary Field Accessors* on page A-3, with the addition that each accessor takes an additional INT argument that specifies which array entry is to be accessed, for example:

```
public voidsetFieldName(int index, FieldType value);
public FieldTypegetFieldName(int index);
```

Array fields specified with the DEPENDING ON clause are handled the same as fixed-size arrays with the following special rules:

1. The accessors may be used to get or set any instance up to the maximum array index.
2. The controlling (DEPENDING ON) variable is evaluated when the DataView is converted to or from an external format, such as a mainframe format. The eGen utility converts only the array elements with subscripts less than the controlling value.

Fields with REDEFINES Clauses

Fields that participate in a REDEFINES set are handled as a unit. A private BYTE[] variable is declared to hold the underlying mainframe data, as well as a private DataView variable. Each of the redefined fields has an accessor or accessors. These accessors take more CPU overhead than the normal accessors because they perform conversions to and from the underlying BYTE[] data.

For example, the copybook:

```
10 MY-RECORD.
20 INPUT-DATA.
30 INPUT-APIC X(4).
30 INPUT-B                PIC X(4).
20 OUTPUT-DATA REDEFINES INPUT-DATA PIC X(8).
```

produces Java code similar to the following:

```
private byte[] m_redef23;
private DataView m_redef23DV;
public InputDataV   getInputData();
public StringgetOutputData();
public voidsetOutputData(String value);
public static class InputDataV extends DataView
{
// Class definition.
}
```

COBOL Data Types

This section summarizes the COBOL data types supported by iWay JAM software. The first table below lists the COBOL data item definitions recognized by the eGen utility. The second table lists the syntactical features and data types recognized by the eGen utility. If a COBOL feature is unsupported and it is not listed as ignored in the table, an error message is generated.

COBOL Feature	Support
IDENTIFICATION DIVISION	Unsupported
ENVIRONMENT DIVISION	Unsupported
DATA DIVISION	Partially Supported
WORKING-STORAGE SECTION	Partially Supported
Data record definition	Supported
PROCEDURE DIVISION	Unsupported
COPY	Unsupported
COPY REPLACING	Unsupported
EJECT, SKIP1, SKIP2, SKIP3	Supported

COBOL Type	Java Type
COMP, COMP-4, BINARY *(integer)*	Short/Int/Long

COBOL Type	**Java Type**
COMP, COMP-4, BINARY *(fixed)*	BigDecimal
COMP-3, PACKED-DECIMAL	BigDecimal
COMP-5	Unsupported
COMP-X	Unsupported
`DISPLAY` *`numeric (zoned)`*	BigDecimal
`BLANK WHEN ZERO` *`(zoned)`*	String
SIGN IS LEADING *(zoned)*	Unsupported
`SIGN IS LEADING SEPARATE` *`(zoned)`*	String
`SIGN IS TRAILING` *`(zoned)`*	String
`SIGN IS TRAILING SEPARATE` *`(zoned)`*	String
edited numeric	String
`COMP-1, COMP-2` *`(float)`*	Unsupported
edited float numeric	String
`DISPLAY` *`(alphanumeric)`*	String
edited alphanumeric	String
INDEX	Int
POINTER	Int
PROCEDURE-POINTER	Unsupported
JUSTIFIED RIGHT	Unsupported (ignored)
SYNCHRONIZED	Unsupported (ignored)
REDEFINES	Supported

COBOL Type	Java Type
66 RENAMES	Unsupported
66 RENAMES THRU	Unsupported
77 level	Supported
88 level *(condition)*	Unsupported (ignored)
group record	Inner Class
`OCCURS` *`(fixed array)`*	Array
`OCCURS DEPENDING` *`(variable-length array)`*	Array
OCCURS INDEXED BY	Unsupported (ignored)
OCCURS KEY IS	Unsupported (ignored)

Other Access Methods for Generated DataView Classes

iWay JAM allows you to access DataView classes through several methods as described in the following topics:

- Mainframe Access to DataView Classes
- XML Access to DataView Classes
- Hashtable Access to DataView Classes

Mainframe Access to DataView Classes

This section describes how mainframe format data may be moved into and out of DataView classes. The eGen Application Generator writes this code for you, so this information is provided as reference.

Mainframe format data may be extracted from a DataView class through the use of the MainframeWriter class. The following example shows a sample of code that may be used to perform the extraction.

```
import com.iwaysoftware.jam.base.io.MainframeWriter;
import com.iwaysoftware.jam.dmd.dataview.DataView;

    ...

    /**
```

```
 * Get mainframe format data from a DataView into a byte[].
 */
byte[] getMainframeData(DataView dv)
{
     try
     {
          MainframeWriter mw = new MainframeWriter();
          // To override the DataView's codepage, change the
          // above constructor call to something like:
          // ...new MainframeWriter("cp1234");

          return dv.toByteArray(mw);
     }
     catch (java.io.IOException e)
     {
          // Some conversion failure occurred…
     }
}
```

If you want to override the codepage provided when the DataView was generated, you may provide another codepage as a String argument to the MainframeWriter constructor, as shown in the comment in the example below.

Loading mainframe data into a DataView is a similar process, in this case requiring the use of the MainframeReader class. The following example shows a sample of code that may be used to perform the load.

```
import com.iwaysoftware.jam.base.io.MainframeReader;
import com.iwaysoftware.jam.dmd.dataview.DataView;

   ...

   /**
    * Put a byte[] containing mainframe format data into a DataView.
    */
   MyDataView     putMainframeData(byte[] buffer)
   {
        MainframeReader mr = new MainframeReader(buffer);
        // To override the DataView's codepage, change the above
        // constructor call to something like:
        // …new MainframeReader("cp1234", buffer);
       .
       .
       .
   MyDataView dv;
       .
       .
       .
   try
        {
         // Construct a new DataView with the mainframe data.
             dv = new MyDataView(mr);

            // Or, to load a pre-existing DataView with mainframe data.
         // dv.mainframeLoad(mr);
        }
        catch (java.io.IOException e)
        {
             // Some conversion failure occurred.
        }

        return dv;
   }
```

XML Access to DataView Classes

Facilities are provided to move XML data into and out of DataView classes. These operations are performed through the use of the XmlLoader and XmlUnloader classes.

- XmlLoader is used to load XML data into a DataView.

- XmlUnloader is used to unload data from a DataView into XML.
- If the eGen script used to produce the DataView specifies the "support xml" option, then both a DTD and an XML/Schema that describe the XML format for this DataView are produced.
- The following example shows an example of the code used to load XML data into a DataView.

```
import com.iwaysoftware.jam.dmd.dataview.DataView;
import com.iwaysoftware.jam.dmd.dataview.XmlLoader;

   ...

   void loadXmlData(String xml, DataView dv)
   {
        XmlLoader xl = new XmlLoader();
        try
        {
             // Load the xml. Note that the xml argument may be either
             // a String or a org.w3c.dom.Element object.
             xl.load(xml, dv);
        }
        catch (Exception e)
        {
             // Some conversion error occurred.
        }
   }
```

The following sample shows an example of the code used to unload a DataView into XML.

```
import com.iwaysoftware.jam.dmd.dataview.DataView;
import com.iwaysoftware.jam.dmd.dataview.XmlUnloader;

   ...

   String unloadXmlData(DataView dv)
   {
        XmlUnloader xu = new XmlUnloader();

        try
        {
             String xml = xu.unload(dv);
             return xml;
        }
        catch (Exception e)
        {
             // Some conversion error occurred.
        }
   }
```

Hashtable Access to DataView Classes

iWay JAM also provides facilities to load and unload DataView objects using Hashtable objects. Hashtable objects are most often used to move data from one DataView to another similar DataView.

When DataView fields are moved into Hashtables, each field is given a key that is a string reflecting the location of the field within the original copybook data structure. The following example shows a sample of a COBOL Copybook.

```
*-------------------------------------------------------
* emprec.cpy
*An employee record.
*-------------------------------------------------------

02emp-record.

04emp-ssn  pic 9(9)  comp-3.

04emp-name.
06emp-name-last  pic x(15).
06emp-name-first  pic x(15).
06emp-name-mi  pic x.

04emp-addr.
06emp-addr-street pic x(30).
06emp-addr-st  pic x(2).
06emp-addr-zip  pic x(9).

* End
```

The fields for the COBOL Copybook in the above example are stored into a Hashtable as shown in the table below.

Key String	Content Type
empRecord.empSsn	BigDecimal
empRecord.empName.empNameLast	String
empRecord.empName.empNameFirst	String
empRecord.empName.empNameMi	String
empRecord.empAddr.empAddrStreet	String
empRecord.empAddr.empAddrSt	String
empRecord.empAddr.empAddrZip	String

Code for Unloading and Loading Hashtables

Following is an example of the code used to **unload** a DataView into a Hashtable.

```
Hashtable ht = new HashtableUnloader().unload(dv);
```

Following is an example of the code used to **load** a Hashtable into an existing DataView.

```
new HashtableLoader().load(dv);
```

Rules for Unloading and Loading Hashtables

The basic rules of Hashtable **unloading** are:

- All data elements in the DataView are placed into the Hashtable.
- Each data item is stored as an object of its Java type. Elements of INT/SHORT/LONG type are converted to Integer/Short/Long.
- Arrays are mentioned at the appropriate level in the key as an index enclosed in "[", "]" pairs. For instance, if empAddr was an array, then one key into the Hashtable might be empRecord.empAddr[2].empAddrStreet.

The basic rules of Hashtable **loading** are:

- All data elements in the DataView attempt to acquire a value from the Hashtable. If no matching key exists, the element retains its original value.
- Hashtable members of the wrong type result in a classCastException being thrown.

Name Translator Interface Facility

A name translator interface facility is available to provide Hashtable name mappings. Both HashtableLoader and HashtableUnloader provide a constructor that accepts an argument of type com.iwaysoftware.jam.dmd.dataview.NameTranslator. The table below lists the descriptions of the public interface methods that must be implemented.

Method	Description
`translate(String input)`	This method received a `String` object as an input parameter and returns a `String` object.

You can write classes that implement this interface for your application. These implementations are used to translate the key strings before the Hashtable is accessed.

Following are some useful implementations that are included in the iWay JAM library:

Class Constructor	Purpose
NameFlattener()	Reduces the key to the portion following the final period character.
PrefixChanger(String old, String add)	Removes an old prefix & adds a new one.

Class Constructor	**Purpose**
PrefixChanger(String old)	Removes a prefix.

The HashtableLoader, HashtableUnloader, and the various name translator classes are included in the "com.iwaysoftware.jam.dmd.dataview" package.

Known Limitations of iWay JAM working with COBOL Copybooks

The following are some of the known limitations of this version of the iWay JAM product.

- Continuation lines are not recognized in COBOL copybooks. This is only a problem for long character literals occurring within VALUES clauses. Comment out the relevant clause to fix the problem.
- COBOL copybooks with array (table) data items having an OCCURS DEPENDING ON clause must be structured so that the depending-on counter data item is not contained within the same group data item as the one containing the array.
- `USAGE` clauses on group data items in COBOL copybooks are not properly propagated to their subordinated member data items.

APPENDIX B

eGen Application Generator Reference

Topics:

This section contains reference pages for the iWay JAM eGen Application Generator (eGen utility). This information includes the rules for writing the script file that controls the code generator.

Synopsis

The eGen utility maps a COBOL copybook into a Java class.

Invoke the utility with the following command:

```
java com.iwaysoftware.jam.egen.EgenCobol scriptfile
```

where:

`java`

is the name of the Java virtual machine executable in the Java Development Kit (JDK).

`com.iwaysoftware.jam.egen.EgenCobol`

is the full class name of the eGen utility.

`scriptfile`

is the script file that controls the eGen utility. You must write this script file on an application-by-application basis. (See the following sample of scriptfile.egen for an example).

If the iWay JAM installation bin directory has been added to your path, the eGen utility may also be invoked with the following command:

```
egencobol scriptfile
```

Listing 0-1 Example of `scriptfile.egen`

```
### example script
#

view demo.CustomDataView from emprec.cpy

service demoService accepts CustomDataView returns CustomDataView

page demoPage "Demo Page"
{
    view demo.CustomDataView

    buttons
    {
    "Try It" service(demoService) shows demoPage
    }
}

servlet demo.DemoServlet shows demoPage
```

Script Syntax Reserved Words

The reserved words shown below must be used as specified in *Grammar* on page B-4.

Note: A reserved word can be used as an identifier if it is enclosed in either single or double quotation marks (refer to *General Rules* on page B-3).

`accepts`	`buttons`	`class`	`client`	`codepage`	`ejb`
`from`	`generate`	`group`	`is`	`method`	`page`
`reset`	`returns`	`server`	`service`	`servlet`	`shows`
`support`	`view`	`transaction`	`xml`		

General Rules

- The `'#'` character and all following characters on the same line are a comment. Use the `'#'` character to specify commented text.
- The character sequence `"//"` and all following characters on the same line are a comment. Use the `"//"` characters to specify commented text.
- The character sequence `"/*"` and all following characters until the occurrence of the sequence `"*/"` are a comment. Use the `"/*"` characters to specify commented text that extends beyond one line.
- White space (including new lines) is not significant, except when it is used to separate tokens. White space includes new lines, carriage returns, tabs, spaces, etc.
- Any sequence of letters, digits, underscores, or periods is a word.
- Any word that does not match a reserved word is an identifier.
- Any sequence of characters is treated as an identifier if it is enclosed in either single or double quotes. This allows the use of reserved words and sequences that contain spaces.

Grammar

The eGen script grammar uses a modified Backus-Naur Form (BNF) syntax, which is used in many industry-standard software reference guides. BNF syntax specifies a context-free grammar. Reserved words are shown in bold. Comments are in italics preceded by a dash (—).

```
script:

definition | script definition

fulldefinition:

generate definition | definition

definition:

viewdef | servicedef | servletdef | ejbdef | classdef | pagedef

viewdef:

view viewname from copybook | viewdf viewmodifier

viewmodifier:

codepage codepagename | support xml

servicedef:

service servicename accepts fullViewname returns fullViewname

servletdef:

servlet classname shows pagename

ejbdef:

clientejb | serverejb

clientejb:

client ejb classname ejbspec { clientmethods }

serverejb:

server ejb classname ejbspec { servermethoddef }

classdef:

client class classname { clientmethods }

ejbspec:

ejbregistration | ejbregistration transactiondef

transactiondef:

transaction [NotSupported | Required | Supports | RequiresNew | Mandatory
| Never]
```

```
pagedef:
page pagename title { view viewname buttons { buttonlist } }
buttonlist:
 buttondef | buttonlist buttondef
buttondef:
servicebutton | ejbbutton
clientmethods:
clientmethoddef | clientmethods clientmethoddef
clientmethoddef:
 method methodname is servicename
servermethoddef:
       method methodname (fullviewname) returns fullviewname
servicebutton:
buttonname service ( servicename ) shows pagename
ejbbutton:
buttonname ejbmethod ( ) shows pagename
viewname:
classname
fullViewname:
viewname | viewname [ codepagename ]
copybook:
identifier
```

An identifier that names a file containing a COBOL data definition.

```
servicename:
identifier
```

An identifier that matches a resource definition.

```
pagename:
identifier
```

An identifier that names a page definition.

`codepagename:`

`identifier`

The name of a codepage to be used for character translation to/from mainframe data formats. This must be a codepage supported by the JDK being used.

`methodname:`

`identifier`

The name to be given to a generated Java method.

`classname:`

`identifier`

An identifier that names a Java class, including any package name.

`ejbregistration:`

`identifier`

The name that will be used to register the home interface for an EJB.

`title:`

`identifier`

The title to be placed into the HTML generated for a page.

`buttonname:`

`identifier`

A button name that will be used in the HTML generated for a page.

`ejbmethod:`

`identifier`

An EJB classname and method specification that should look like this:

`package.ejbclass.method`

or

`ejbclass.method`

Results of Running the eGen Application Generator

- The specified COBOL copybook is parsed for each DataView definition (described in Appendix A, *DataView Programming Reference*) and a Java source file for the specified DataView class is generated in the current directory.

  If XML support was requested, then the following files are also produced:

  `viewname.dtd` - DTD file

  `viewname.xsd` - XML Schema file

- For each servlet definition, a Java source file is generated in the current directory for the specified class.
- For each client class definition, a Java source file is generated in the current directory for the specified class.
- For each EJB definition, three Java source files, a WebLogic deployment information file, and a deployment descriptor text file are generated in the current directory. The names of the generated files are listed in below.

Name of File	**Purpose**
`*classname*Home.java`	EJB Home Interface
`*classname*Bean.java`	EJB Implementation class
`*classname*.java`	EJB Remote Interface
`*classname*-jar.xml`	EJB Deployment descriptor
`wl-classname-jar.xml`	WebLogic Deployment Info

APPENDIX C

Understanding How iWay JAM Uses XML

Topic:

iWay Java Adapter for Mainframe (iWay JAM) uses the capabilities of XML to exchange data between different applications and operating systems. Understanding basic XML terms will help you to understand iWay JAM XML capabilities and how they are used.

What is XML?

Extensible Markup Language, or XML, is a text format for exchanging data between different systems. It allows data to be described in a simple, standard, text-only format. Since the data is presented in a standard form, applications on disparate systems can interpret the data using simple text parsing tools, instead of having to interpret data in proprietary binary formats.

XML documents come in two varieties: data and metadata.

- XML Data Document

  Data records can be converted into XML documents, which can then be transmitted to other applications. The XML data documents contain a single top-level entity (or tag) that represents the entire data record. Fields within the record are represented by other subordinate entities nested within the top-level entity. Each entity has a unique tag name, which corresponds to a field within the original data record. Each entity has content, which is the actual data value of the field. Entities may also have attributes, which are values attached to the entities that augment the normal content values. The XML data document file name ends with a .xml extension.

See Listing for an example XML data document.

- XML Metadata

  Every XML document consists of a top-level entity, which in turn may be composed of subordinate entities. The structure of these entities, which included their tag names, the order in which they occur, the type and length of their content values, and their allowed attribute values, is described by a metadata definition. Metadata definitions can take the form of XML documents themselves. There are two standard formats for XML metadata documents: XML Document Type Definition (DTD) and XML Schema.

Document Type Definition

A Document Type Definition, or DTD, defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements (tags). While XML provides an application independent way of sharing data, the DTD provides a common definition for interchanging data.

Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

The XML DTD file name ends with a .dtd extension.

See Listing 0-2 for an example XML DTD document.

XML Schema

A schema specifies the structure of an XML document and constraints on its content. While XML is the meta-language that provides the rules for defining tag languages, an XML Schema document is a formal specification of the grammar for a particular tag language. The schema defines the elements that can appear within the document and the attributes that can be associated with an element. It also defines the structure of the document: which elements are child elements of others, the sequence in which the child elements can appear, and the number of child elements. It defines whether an element is empty or can include text. The schema can also define default values for attributes.

XML Schema is more precise than DTD, providing more descriptive information about each XML element. It is likely that XML Schema will eventually replace XML DTD as the dominant standard metadata format.

A schema is useful for validating the document content to determine whether a document is a valid instance of the grammar expressed by that schema and for describing your grammar for use by others.

The XML Schema file name ends with a .xsd extension.

See Listing 0-3 for an example XML Schema document.

How iWay JAM Uses XML

The iWay JAM eGen Application Generator provides the ability to generate both XML Schema and XML DTD (Document Type Definition) documents for a given COBOL copybook record definition. The iWay JAM runtime environment provides the capability of converting data records into XML data documents formatted according to their corresponding schema or DTD definitions.

The following listings show examples of the XML files generated by the eGen utility from the COBOL Copybook for an employee information record.

Listing 0-1 shows an example of an employee information record from a COBOL Copybook. The eGen utility generates an XML Schema and a DTD from the employee information record. Listing shows the corresponding XML document that conforms to the XML Schema and DTD generated from the employee record information, Listing 0-2 shows the corresponding DTD, and Listing 0-3 shows the corresponding XML Schema.

Listing 0-1 COBOL Copybook for Employee Information Record (emprec.cpy)

```
*--------------------------------------------------------
* emprec.cpy
* Employee record.
*
* @(#)$Id: emprec.cpy,v 1.2 1999/11/12 01:16:41 $                 *------
-------------------------------------------------
        02  emp-record.

        04  emp-ssn                     pic 9(9) comp-3.

        04  emp-name.
                06  emp-name-last       pic x(15).
                06  emp-name-first      pic x(15).
                06  emp-name-mi         pic x.

        04  emp-addr.
                06  emp-addr-street     pic x(30).
                06  emp-addr-st         pic x(2).
                06  emp-addr-zip        pic x(9).

* End
```

Example XML Document that Conforms to a DTD and XML Schema Generated from the eGen Application Generator (emprec.xml)

```
<emprec>
   <empRecord>
     <empSsn>660337645</empSsn>
     <empName>
       <empNameLast>Doe</empNameLast>
       <empNameFirst>John</empNameFirst>
       <empNameMi>P</empNameMi>
     </empName>
     <empAddr>
       <empAddrStreet>3235 Possum Park Ln.</empAddrStreet>
       <empAddrSt>TX</empAddrSt>
       <empAddrZip>758050000</empAddrZip>
     </empAddr>
   </empRecord>
</emprec>
```

Listing 0-2   DTD Generated from eGen Application Generator (emprec.dtd)

```
<!--
! DTD emprec 1.0
!
! Definition:    emprec
! Version:       1.0
! Source:        ../symbol/emprec.cpy
! Generated:     2000-09-27T19:18:25.084Z
! Created:       2000-09-27T19:18:24.937Z
! Modified:      1999-11-12T01:16:41.000Z
!-->

<!ELEMENT emprec
 ( empRecord )>

<!ATTLIST emprec
  date CDATA #DEFAULT "unknown">
  <!-- format="ccyy-mm-ddThh:mm:ss.mmmZ" -->

<!ATTLIST emprec
  version CDATA #DEFAULT "1.0">

<!-- empRecord -->
<!ELEMENT empRecord
 ( empSsn ,
   empName ,
   empAddr )>


<!-- empRecord.empSsn -->
<!ELEMENT empSsn
  (#PCDATA)>

<!-- empRecord.empName -->
<!ELEMENT empName
 ( empNameLast ,
   empNameFirst ,
   empNameMi )>

<!-- empRecord.empName.empNameLast -->
<!ELEMENT empNameLast
  (#PCDATA)>

<!-- empRecord.empName.empNameFirst -->
<!ELEMENT empNameFirst
  (#PCDATA)>
```

```
<!-- empRecord.empName.empNameMi -->
<!ELEMENT empNameMi
  (#PCDATA)>

<!-- empRecord.empAddr -->
<!ELEMENT empAddr
 ( empAddrStreet ,
   empAddrSt ,
   empAddrZip )>

<!-- empRecord.empAddr.empAddrStreet -->
<!ELEMENT empAddrStreet
  (#PCDATA)>

<!-- empRecord.empAddr.empAddrSt -->
<!ELEMENT empAddrSt
  (#PCDATA)>

<!-- empRecord.empAddr.empAddrZip -->
<!ELEMENT empAddrZip
  (#PCDATA)>

<!-- End -->
```

Listing 0-3 XML Schema Generated from eGen Application Generator (emprec.xsd)

```
<?xml version="1.0"?>
<schema
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:annotation>
    <xsd:documentation>
      Schema:       emprec
      Version:      1.0
      Source:       ../symbol/emprec.cpy
      Generated:    2000-09-27T19:19:42.857Z
      Created:      2000-09-27T19:19:43.708Z
      Modified:     1999-11-12T01:16:41.000Z
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="emprec">
    <xsd:complexType>

      <xsd:attribute name="date"
         type="xsd:timeInstant"/>

      <xsd:attribute name="version"
         type="xsd:string"
         use="default"
         value="1.0"/>

      <xsd:element name="empRecord">
        <xsd:complexType>

          <xsd:element name="empSsn">
            <xsd:simpleType base="xsd:integer">
              <xsd:precision value="9"/>
              <xsd:minInclusive value="0">
            </xsd:simpleType>
            <!-- <%picture value="9(9)"/> -->
          </xsd:element>

          <xsd:element name="empName">
            <xsd:complexType>

              <xsd:element name="empNameLast"
                 type="xsd:string"
                 length="15"/>
                 <!-- <%picture value="x(15)"/> -->

              <xsd:element name="empNameFirst"
                 type="xsd:string"
```

```
                  length="15"/>
                  <!-- <%picture value="x(15)"/> -->

               <xsd:element name="empNameMi"
                  type="xsd:string"
                  length="1"/>
                  <!-- <%picture value="x"/> -->

             </xsd:complexType>
           </xsd:element> <!--"empName"-->

           <xsd:element name="empAddr">
             <xsd:complexType>

               <xsd:element name="empAddrStreet"
                  type="xsd:string"
                  length="30"/>
                  <!-- <%picture value="x(30)"/> -->

               <xsd:element name="empAddrSt"
                  type="xsd:string"
                  length="2"/>
                  <!-- <%picture value="x(2)"/> -->

               <xsd:element name="empAddrZip"
                  type="xsd:string"
                  length="9"/>
                  <!-- <%picture value="x(9)"/> -->

             </xsd:complexType>
           </xsd:element> <!--"empAddr"-->

         </xsd:complexType>
       </xsd:element> <!--"empRecord"-->


     </xsd:complexType>
   </xsd:element> <!--"emprec"-->

 </schema>
```

APPENDIX D

RMI Access to the iWay JAM Gateway

Topics:

A Remote Method Interface (RMI) configuration subsystem allows you to monitor and control the iWay JAM Gateway by a remote Java application. iWay JAM provides such administrative capabilities through the com.iwaysoftware.jam.Admin utility. Using the features in the RMI subsystem, referred to as the JAM Deployed Configuration feature, you can develop your own custom administrative capabilities.

JAM Deployed Configuration Feature

The JAM Deployed Configuration feature is comprised of several of RMI-based interfaces which are advertised in the JNDI tree of a WebLogic Server hosting the iWay JAM Gateway. These objects are constructed, at server boot time, based on the iWay JAM configuration information specified in the Administration Server of the iWay JAM domain. Information about the CRM, Links, and Services is maintained in the jamconfig.xml file. Update this file using the WebLogic Administration Console.

Note: For information about updating the jamconfig.xml file, refer to the *iWay Java Adapter for Mainframe Configuration and Administration Guide*.

The following objects are provided for remote gateway access:

- GatewayBootstrap

  Primary access point to the Gateway(s) configured for a given WebLogic Server. This object allows access to Gateways without requiring knowledge of the Gateway name.

- DeployedGateway

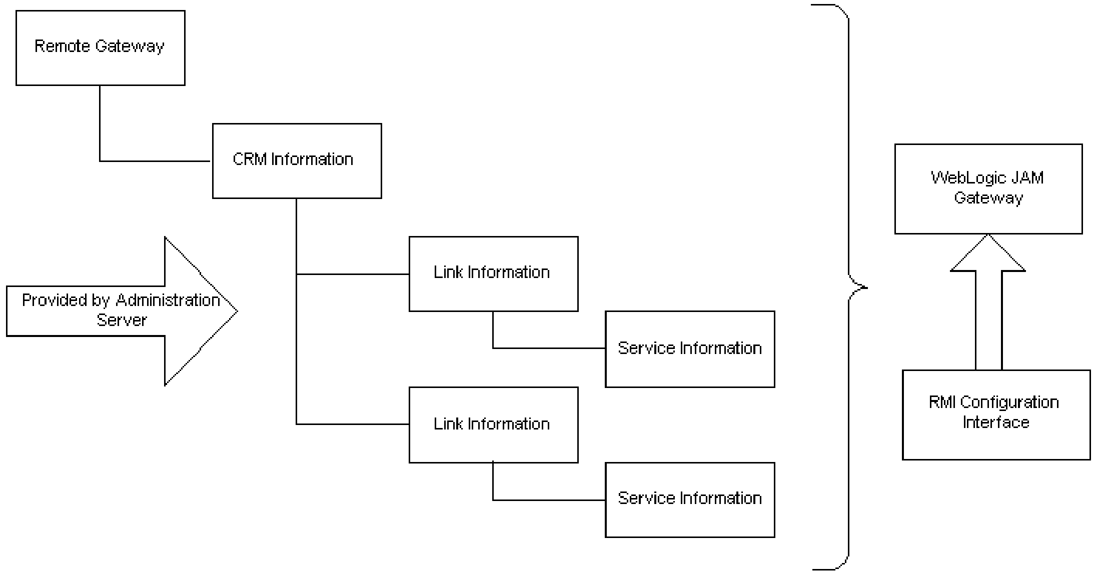
  This object provides access to all the remote functionality of the iWay JAM Gateway. It permits statistics to be gathered, starting/stopping the gateway, and returns information about deployed CRMs, Links, and services. If the Gateway name is known this object may be obtained directly from JNDI.

  The following objects are obtained via the DeployedGateway object and provide additional information about the configuration and status of iWay JAM:

Object	**Description**
DeployedCRM	Provides a read-only copy of the CRM configuration being used by the Gateway.
DeployedLink	Provides read-only access to information about a CRM link.
DeployedService • DeployedSession (object obtained using a DeployedService Object)	DeployedService contains information about an iWay JAM service. DeployedSession is a remote implementation of an iWay JAM session, which may be used to invoke the associated mainframe service. **Note:** While this capability may be useful to customers at times, it is strongly recommended that the EgenClient class be used for service invocation.

Object	Description
ActivityCounts	Provides count information for the respective Gateway/ service. These counts reflect, for example, the number of requests processed by the Gateway, average response time, and number of failures.

These remote objects are arranged in a hierarchy within the WebLogic Server node as follows:



GatewayBootstrap Object

The GatewayBootstrap object is bound into the JNDI tree of each WebLogic Server hosting an iWay JAM Gateway. There is a single GatewayBootstrap object per WebLogic Server ("pinned" object) and its JNDI name, com.iwaysoftware.jam.bootstrap, is the same on all server instances. This JNDI name is available via the named constant GatewayBootstrap.JNDI_NAME to eliminate literal hardcoding.

The following code listing demonstrates obtaining the GatewayBootstrap object from a WebLogic Server accessible via the URL t3://dynamo1:7001.

```
Properties prop = new Properties();
prop.setProperty(Context.INITIAL_CONTEXT_FACTORY,
     weblogic.jndi.WLInitialContextFactory.class.getName());
prop.setProperty(Context.PROVIDER_URL, "t3://dynamo1:7001");
prop.setProperty(Context.SECURITY_PRINCIPAL, "UserId");
prop.setProperty(Context.SECURITY_CREDENTIALS, "Password");

Context ctx = new InitialContext(prop);
GatewayBootstrap gwBoot;
gwBoot = (GatewayBootstrap) ctx.lookup(GatewayBootstrap.JNDI_NAME);
```

This example obtains a JNDI Initial Context to the server in question and then looks up the GatewayBootstrap object using the predefined constant object name.

Following are the methods offered by the GatewayBootstrap object:

Reference: com.iwaysoftware.jam.cluster.GatewayBootstrap

Implements java.rmi.Remote

JAM Gateway bootstrap class is used to allow the creation and/or retrieval of Gateways on WebLogic Server nodes. This object is a singleton per WebLogic Server node and is registered in the JNDI tree as a local (i.e. REPLICATE_BINDINGS=false) object. It is created by the iWay JAM server startup task and remains available for the life of the server.

Method

The following methods are available with the com.iwaysoftware.jam.cluster.GatewayBootstrap object to retrieve Gateway information:

Method	Description
com.iwaysoftware.jam.cluster.DeployedGateway getGateway(String gatewayName)	Returns an iWay JAM gateway existing on the local WebLogic Server node. • Returns: a DeployedGateway remote object. If the gateway does not exist on the local node a null is returned. • Throws: RemoteException if a communication error is encountered.

Method	Description
com.iwaysoftware.jam.cluster.DeployedGateway[] getGateways()	Returns an array of the iWay JAM Gateways existing on the local WebLogic Server node. • Returns: an array of DeployedGateway remote objects. If no gateways exist on the local node an empty array is returned. • Throws: RemoteException if a communication error is encountered.

Fields

public static final JNDI_NAME

DeployedGateway Object

The DeployedGateway object is bound into the WebLogic JNDI tree using a name that is constructed of a literal prefix, DeployedGateway.JNDI_PREFIX, and suffixed by the name of the iWay JAM Gateway represented by the object. For example, the JNDI name of an iWay JAM Gateway named MyJAM would be:

```
DeployedGateway.JNDI_PREFIX+"MyJAM".
```

The DeployedGateway object for an iWay JAM Gateway may be obtained either from a GatewayBootstrap object or directly via a JNDI lookup.

The following code listing demonstrate three different ways of obtaining a DeployedGateway object named JAM1 from WebLogic Server t3://dynamo1:7001.

- Obtain the DeployedGateway using the GatewayBootstrap object retrieved by the sample code:

  ```
  DeployedGateway gw = gwBoot.getGateway("JAM1");
  ```

- Obtain the DeployedGateway using the GatewayBootstrap object retrieved by the previous sample code. This example assumes JAM1 is the only gateway available and retrieves the remote object anonymously:

  ```
  DeployedGateway gw = gwBoot.getGateways()[0];
  ```

- Obtain the DeployedGateway directly from the JNDI tree:

```
Properties prop = new Properties();
prop.setProperty(Context.INITIAL_CONTEXT_FACTORY,
     weblogic.jndi.WLInitialContextFactory.class.getName());
prop.setProperty(Context.PROVIDER_URL, "t3://dynamo1:7001");
prop.setProperty(Context.SECURITY_PRINCIPAL, "UserId");
prop.setProperty(Context.SECURITY_CREDENTIALS, "Password");

Context ctx = new InitialContext(prop);
DeployedGateway gw = (DeployedGateway) ctx.lookup(
                     DeployedGateway.JNDI_PREFIX + "JAM1");
```

Following are the methods offered by the DeployedGateway object:

Reference: com.iwaysoftware.jam.cluster.DeployedGateway

Implements java.rmi.Remote

This is the remote interface for the deployed gateway object. One DeployedGateway object exists for each configured iWay JAM Gateway. The DeployedGateway is placed in the JNDI tree under the name com.iwaysoftware.jam.gateway. GatewayName where GatewayName is the configured name of the Gateway.

Method

The following methods are available with the com.iwaysoftware.jam.cluster.DeployedGateway object to retrieve Gateway information:

Method	Description
boolean isEnabled()	Returns a value indicating if this iWay JAM Gateway is deployed and running. • Throws: RemoteException if a communication error is encountered.
boolean isStarting()	Returns a value indicating if this iWay JAM Gateway is starting up. • Throws: RemoteException if a communication error is encountered.

Method	Description
boolean isStopped()	Returns a value indicating if this iWay JAM Gateway is stopped. • Throws: RemoteException if a communication error is encountered.
boolean isDeployed()	Returns a value indicating if this iWay JAM Gateway is deployed.
java.lang.String getName()	Retrieves the name of this Gateway. • Throws: RemoteException if a communication error is encountered.
com.iwaysoftware.jam.cluster.DeployedCRM getCRM()	Retrieves information on the CRM used by this Gateway. • Throws: RemoteException if a communication error is encountered.
void setStopWlsOnExit(boolean wlsStopOnExit)	Returns a boolean value indicating whether to terminate WebLogic Server if the iWay JAM Gateway shuts down. If set to true, WebLogic Server will terminate when the Gateway shuts down.
com.iwaysoftware.jam.cluster.DeployedLink getLink(String linkName)	Retrieves a DeployedLink object representing the requested link on this Gateway. • Parameter: linkName - The name of the link to retrieve. • Throws: DeployedException if the linkName is invalid or does not exist on this Gateway. RemoteException if a communication error is encountered.

Method	Description
com.iwaysoftware.jam.cluster.DeployedLink[] getLinks()	Retrieves an array of DeployedLink objects representing all the links defined on the CRM used by this Gateway. • Throws: RemoteException if a communication error is encountered.
com.iwaysoftware.jam.cluster.DeployedService getService(String serviceName)	Retrieves a DeployedService stub for the named service object. • Parameter: serviceName - The name of the service to retrieve. • Throws: DeployedException if the serviceName is invalid or does not exist on this Gateway. RemoteException if a communication error is encountered.
v**oid deploy(boolean deployed)**	Marks this Gateway as being deployed or undeployed. If an enabled Gateway is undeployed it will be shut down. • Parameter: deployed - true if the Gateway is deployed and false otherwise. • Throws: RemoteException if a communication error is encountered.
void enableLink(String linkName)	Enables all services being offered by link linkName. • Throws: DeployedException if the named link is unknown. RemoteException if a communication error is encountered.

Method	Description
void disableLink(String linkName)	Disables all services being offered by link linkName. • Parameter: linkName - The name of the link to be undeployed. • Throws: DeployedException if the named link is unknown. RemoteException if a communication error is encountered.
com.iwaysoftware.jam.cluster.Activity Counts getServiceActivity(String serviceName)	Gets the activity counts for this service. • Throws: RemoteException if a communication error is encountered.
com.iwaysoftware.jam.cluster.Activity Counts getLocalServiceActivity(String serviceName)	Gets the activity counts for this local service. • Throws: RemoteException if a communication error is encountered.
void startup()	Starts this iWay JAM Gateway. If the Gateway is already running, this method simply returns. • Throws: RemoteException if a communication error is encountered.
void shutdown()	Stops this iWay JAM Gateway. If the Gateway is already stopped, this method simply returns • Throws: RemoteException if a communication error is encountered.

Method	Description
com.iwaysoftware.jam.cluster.ActivityCounts[] getGatewayActivity()	Gets the activity counts for all services offered by this Gateway. • Throws: RemoteException if a communication error is encountered.

Fields

public static final JNDI_PREFIX

JNDI prefix used by all DeployedGateway objects.

DeployedCRM Object

The DeployedCRM object is obtained via the DeployedGateway.getCRM() method. This object is a read-only wrapper providing the information used to configure the CRM being used by this iWay JAM Gateway.

Reference: com.iwaysoftware.jam.cluster.DeployedCRM

Implements java.io.Serializable

Method

The following methods are available with the com.iwaysoftware.jam.cluster.DeployedCRM object.

Method	Description
boolean isEnabled()	Determines if this CRM is enabled.
java.lang.String getName()	Retrieves the name of this CRM.
java.lang.String getListenAddress()	Retrieves the host NIC address where this CRM listens for connections.
int getListenPort()	Retrieves the host port where this CRM listens for connections.
java.lang.String getLU()	Retrieves the APPC Logical Unit used for this CRM.
java.lang.String getStackType()	Retrieves the type of stack used by this CRM for CICS/IMS communication.

DeployedLink Object

The DeployedLink object is obtained via the DeployedGateway.getLink() or DeployedGateway.getLinks() methods. It is a read-only wrapper providing information about a particular CRM mainframe link. Following are the methods offered by this object:

Reference: com.iwaysoftware.jam.cluster.DeployedLink

Implements java.io.Serializable

This is the interface for the deployed link object. DeployedLink objects are created by their parent Gateways and are used to the services offered on each Gateway link.

Method

The following methods are available with the com.iwaysoftware.jam.cluster.DeployedLink object.

Method	Description
boolean isEnabled()	Determines if this link is enabled on any gateway on the local WLS node.
java.lang.String getName()	Retrieve the name of this link.

DeployedService Object

The DeployedService object represents an iWay JAM outbound service currently being offered by a CRM mainframe region link. The DeployedService object may be obtained via the DeployedGateway.getService() method. It may also be obtained directly via a JNDI lookup using a literal prefix and the name of the service as a key. The following code listing demonstrates obtaining the DeployedService object for an outbound service named NewEmployee.

```
Properties prop = new Properties();
prop.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    weblogic.jndi.WLInitialContextFactory.class.getName());
prop.setProperty(Context.PROVIDER_URL, "t3://dynamo1:7001");
prop.setProperty(Context.SECURITY_PRINCIPAL, "UserId");
prop.setProperty(Context.SECURITY_CREDENTIALS, "Password");

Context ctx = new InitialContext(prop);
DeployedService svc = (DeployedService) ctx.lookup(
                DeployedService.JNDI_PREFIX + "NewEmployee");
```

Following are the methods offered by the DeployedService object:

Reference: com.iwaysoftware.jam.cluster.DeployedService

Implements java.rmi.Remote

This is the remote interface for the deployed service object. There is one service object defined on each WebLogic Server node for each unique service name. The service object is responsible for the following:

- maintaining information on the local Gateways and links offering the service
- load balancing between the Gateways on a single WebLogic Server

Method

The following methods are available with the com.iwaysoftware.jam.cluster.DeployedService object.

Method	Description
boolean isEnabled()	Determines if this service is enabled on any link serviced by a Gateway on the local WebLogic Server node. • Throws: RemoteException if a communication error is encountered.
void deploy(boolean deployed)	Sets the deployment state of this service. If the service is undeployed while it is enabled it will be disabled as part of the undeployment. • Parameter: deployed - Pass as true if the service is deployed and false otherwise. • Throws: RemoteException if a communication error is encountered.
java.lang.String getName()	Retrieves the name of this service. • Throws: RemoteException if a communication error is encountered.

Method	Description
com.iwaysoftware.jam.cluster.DeployedGateway[] getGateways()	Retrieves an array of the local Gateways offering this service. • Throws: RemoteException if a communication error is encountered.
com.iwaysoftware.jam.cluster.DeployedLink[] getLinks(String gatewayName)	Retrieves an array of links offering this service for the specified gateway. • Parameter: gatewayName - The name of the Gateway for which links are to be returned. If this parameter is passed as null, all links offering the service on the local WebLogic Server node will be returned. • Throws: RemoteException if a communication error is encountered.
com.iwaysoftware.jam.cluster.DeployedSession getSession()	Returns a session object that may be used to invoke this service on the mainframe via the iWay JAM Gateway. • Throws: RemoteException if a communication error is encountered.

Fields

public static final JNDI_PREFIX

JNDI prefix used by all replicated DeployedService objects.

ActivityCounts Object

The ActivityCounts object contains a number of long integer fields representing various statistics for an iWay JAM service. This object may be obtained for a particular service via the DeployedService.getServiceActivity() or DeployedGateway.getLocalServiceActivity() methods. In addition, the ActivityCount objects for all services offered by a Gateway may be obtained via the DeployedGateway. getGatewayActivity() method.

Following are the data fields available from the ActivityCounts object:

Reference: com.iwaysoftware.jam.cluster.ActivityCounts

Implements java.io.Serializable

Statistics container. Holds a set of activity counts.

Constructor Fields

public ActivityCounts()

public m_requests

The number of requests that have reached the gateway. This may be larger than the sum of successes and failures if some requests are still being processed.

public m_successes

The number of requests that have successfully been processed to completion by the gateway. Application level failures may be reported as gateway successes.

public m_failures

The total number of failures of any kind.

public m_timeouts

The number of requests that have timed out.

public m_lateReplies

The number of requests that timed out and then received a response. These will also be included in m_timeouts.

public m_timedRequestCount

The number of requests accounted for in the m_totalResponseTime field.

public m_totalResponseTime

The sum of the response times for all the requests that have reached the network.

public m_minResponseTime

The shortest response time registered during this Gateway session.

public m_maxResponseTime

The longest response time registered during this Gateway session.

DeployedSession Object

The DeployedSession object may be obtained via a DeployedService remote object. It represents an iWay JAM session which may be used to invoke the associated service on the mainframe. Following are the methods offered by this object:

Reference: com.iwaysoftware.jam.cluster.DeployedSession

Implements java.rmi.Remote

This is the remote interface for the deployed session object. This object wraps a jcrmSession object allowing remote access via the DeployedService object. This object is not bound into the JNDI tree and is not clusterable. A DeployedSession object is obtained by calling the DeployedService.getSession() method.

See Also: **DeployedService**

Method

The following methods are available with the com.iwaysoftware.jam.cluster.DeployedSession object.

Method	Description
void setUser(T3User user)	Sets the userid and password on the remote session object. This method is preferred over setUserid and setPassword as it encrypts the credential data. • Parameter: user - A weblogic.common.T3User object containing the userid and password to be used for this iWay JAM session. • Throws: RemoteException if a communication error is encountered.

Method	Description
void setUserid(String u)	Sets the userid to be used by the iWay JAM session. This method will transmit the userid in clear text assuming that the remote session object is not co-located. Setting user information via the setUser method is preferred for security reasons. • Parameter: u - The userid to be used for the JAM session. • Throws: RemoteException if a communication error is encountered. • See Also: **setUser(T3User user)**
void setPassword(String p)	Sets the password to be used by the iWay JAM session. This method will transmit the password in clear text assuming that the remote session object is not co-located. Setting user information via the setUser method is preferred for security reasons. • Throws: RemoteException if a communication error is encountered. • See Also: **setUser(T3User user)**

Method	Description
void dispatch()	Dispatches the iWay JAM service request to the mainframe. • Throws: RemoteException in the event of a communication error. If an SNA error is encountered while communicating with the mainframe an snaException will be nested in the RemoteException.
java.lang.String getServiceName()	Returns the name of the service this session was established to invoke. • Throws: RemoteException if a communication error is encountered.
byte[] getDataBuffer()	Retrieves the data buffer returned by a successful service invocation. • Throws: RemoteException if a communication error is encountered.
boolean setDataBuffer(byte[] buffer)	Sets the data buffer to be used as input the the mainframe service. This method must be called prior to dispatching the service request. • Parameter: buffer - A byte array containing the buffer to be used as input to the mainframe service. • Returns: A boolean value indicating if the buffer was successfully set. • Throws: RemoteException if a communication error is encountered.

Method	Description
byte[] runService(T3User user, byte[] buffer)	Sets the user/password and data buffer, dispatches the service call and returns the response buffer. This method may be used to reduce the network round trips in running a remote service. • Parameter: user - A weblogic.common.T3User object containing the userid and password to be used for this iWay JAM session. buffer - A byte array containing the buffer to be used as input to the mainframe service. • Returns: A byte array containing the buffer returned by the mainframe service. • Throws: RemoteException in the event of a communication error. If an SNA error is encountered while communicating with the mainframe an snaException will be nested in the RemoteException.
void close()	Closes this session object and returns it to the iWay JAM session pool. This method should be called after dispatching a service and retrieving the resulting data buffer to free up resources and allow the session to be allocated to other users. • Throws: RemoteException if a communication error is encountered.

Reader Comments

In an ongoing effort to produce effective documentation, the Documentation Services staff at Information Builders welcomes any opinion you can offer regarding this manual.

Please use this form to relay suggestions for improving this publication or to alert us to corrections. Identify specific pages where applicable. You can contact us through the following methods:

Mail:	Documentation Services - Customer Support Information Builders, Inc. Two Penn Plaza New York, NY 10121-2898
Fax:	(212) 967-0460
E-mail:	books_info@ibi.com
Web form:	http://www.informationbuilders.com/bookstore/derf.html

Name:____

Company:____

Address:____

Telephone:_____________________________Date:_______________________

E-mail:____

Comments:

Reader Comments

Information Builders, Two Penn Plaza, New York, NY 10121-2898 **(212) 736-4433**

iWay Java Adapter for Mainframe Programming Guide DN3501654.0207
Version 5.1 Service Pack 1