

JI Integration

Client Developer's Guide

Version 4.5

June 2025
(originally released December 2004)

This document applies to JI Integration Version 4.5 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 1999–2025 Software GmbH, Darmstadt, Germany and/or their suppliers. All rights reserved.

The name Software GmbH and all Software GmbH product names are either trademarks or registered trademarks of Software GmbH. Other company and product names mentioned herein may be trademarks of their respective owners.

Document ID: JI-CDG-45-20250131

Table of Contents

About this Guide 3

Formatting Conventions	4
Documentation Set	5
Viewing the Documentation Online	6

Chapter 1. Overview7

The JI Integration Environment	7
The Middleware Model	7
Host Communications	8
Client Libraries and Interfaces	8
About JI Integration Clients	8
Client/Service Interaction: An Overview	9
Service Sessions	10
JI Integration Client Libraries and Interfaces	10
SOCKS Functionality	10
SOCKS Overview	10
Integration Architecture	11
Currency	11
Client XML Support	12
XML Support Overview	12
String Size Restrictions	12
Client Input Message - Specifying XML Parsing	12
Errors Encountered During Parsing	13
MapMaker Generated DTD	13
Parsing Rules	14
Overview	14
Configurable Parsing Options	14
Attribute Parsing Options	15
Returning XML Output to the Client	17
JavaServer Pages (JSP) Support	18
Active Server Pages (ASP.NET) Support	21

Chapter 2. The Java Client Library Version 3 (JClient3)23

Using the JClient3	23
Deploying Clients Developed with the JClient3 Library	23
Package com.jacada.ea.jclient3	24
Package com.jacada.ea.jclient3.utility	24
Creating a Connection to an Environment Manager	24
EnvironmentManagerConnectionProperties Class	24
EnvironmentManagerConnection Class	25
Invoking a Method and sending Input Data to the Service	25
Closing the Service Connection	26

Using Threads with the JClient3 Library	26
Supported DataTypes	26
The java.util.List Datatype	27
The java.util.Map Datatype	28
SOCKS Functionality	29
Java Client Changes	29
SOCKS Configuration File	29
Creating a Terminal Window	30
Information Required to use Terminal Windows	31
JavaBean Support	33
Constraints on Implementations	33
Metadata	34
Java Archive Files	34
UNIX Bourne Shell Example	34
Windows Example	34
Example Code	35
Example 1: Echo Test	35
Example 2: Protocol Agent Terminal Window Test	42
Example Three: Metadata Query Test	49
 Glossary	57
 Index	1

About this Guide

Welcome to the *JI Integration Client Developer's Guide*.

Before You Begin

This guide is intended for software developers who want to create client applications using JI Integration.

In order to use JI Integration, developers should have a working knowledge of the following:

- JI Integration services.
- UNIX and/or Microsoft® Windows®.
- The location and communication requirements of existing legacy data and applications.

Optional requirements include:

- Java programming language for developing Java-based JI Integration clients.
- C programming language for developing C-based JI Integration clients.
- Siebel Tools and Siebel Enterprise Relationship Management applications to integrate JI Integration with Siebel applications.

Organization

This guide is organized as follows:

- “Overview” on page 7 introduces JI Integration client libraries and interfaces, and provides a high-level overview of these clients are used with JI Integration.
- “The Java Client Library Version 3 (JClient3)” on page 23 describes how to use the Java Client Library version 3 (JClient3) to develop JI Integration clients using the Java programming language.
- “Glossary” on page 57.
- “Index” on page 1.

Formatting Conventions

The following formatting conventions are used in this manual:

Table 1. Formatting conventions

Convention	Used for..
<i>Italics</i>	Italics are used for files, directories, programs, and book titles. For example: <code><JI_install_dir>/bin/ea_mapmaker.exe</code> .
Monotype font	A monotype font is used to represent examples of code, characters that the user enters, and prompts or messages from the system. For example: Type <code>ea_start</code> at the command prompt.
Sans-serif font	A sans-serif font is used to represent Graphical User Interface (GUI) features, such as buttons. For example: Press the Help button to display a list of help topics.
Serif bold font	This font is used for notes and warnings that require special attention. For example: Warning: You must install JI Integration in an empty directory.

Documentation Set

JI Integration is supplied with the manuals shown below. The documentation is delivered in Adobe Acrobat Reader Portable Document Format (PDF). No hardcopy documentation is provided, but you can print the PDF files on your local printer.

Use this guide in conjunction with other manuals provided with JI Integration:

Table 2. JI Integration documentation set

Title	Description
<i>JI Integration Release Notes</i>	Provides information about additions and revisions to the current release of JI Integration. The release notes are distributed in two forms, as a PDF and as a text file
<i>JI Integration Installation and Configuration Guide</i>	Details installation procedures for JI Integration.
<i>JI Integration Tutorial</i>	Provides hands-on instruction about how to write JI Integration services using MapMaker
<i>JI Integration User's Guide</i>	Describes how to configure the JI Integration environment, as well as how to use JI Integration graphical development and system monitoring tools.
<i>JI Integration Client Developer's Guide</i>	Describes how to design and develop JI Integration client applications, as well as how to integrate JI Integration services into third-party development environments.
<i>JI Integration Integration Guide</i>	Contains information about integrating JI Integration Services with other technologies, such as Siebel eBusiness Applications, MQSeries, Web Services, and more.

Table 2. JI Integration documentation set

Title	Description
<i>JI Integration Supplemental Reference Guide</i>	Contains additional information on JI Integration commands, error codes, language translation, keyboard mapping, logging, licensing, file formats, and field attributes. This guide replaces the Appendices that were duplicated across the manual set.

Viewing the Documentation Online

Online documentation is available in the following locations:

- In JI Integration Graphical User Interfaces (GUIs), click on the **Help** menu and select **Help Topics** to display the JI Integration documentation.
- JI Integration documentation is available on-line in Adobe® Acrobat™'s Portable Document Format (PDF). If the documentation has been installed, open the file <JI_install_dir>/doc/_ji_doc.pdf in Adobe Acrobat Reader™ or a Web browser (where <JI_install_dir> is the location of your JI Integration installation).

You can also access the latest version of the documentation for Software GmbH products at <http://documentation.softwareag.com/>. As new versions become available, the documentation on this web site will be updated and the previous versions will be migrated to the Empower Product Support Web site at <https://empower.softwareag.com/>. If you have a maintenance contract, you can view all versions of documentation on this web site. You will find instructions for registering and obtaining a userid and password on the documentation web site.

Chapter 1. Overview

This chapter provides an introduction to JI Integration client libraries and interfaces, and describes how JI Integration clients interact with the JI Integration environment and with JI Integration services.

The JI Integration Environment

The Middleware Model

JI Integration is a middleware solution providing real-time access and interaction between front-end user environments, often called “clients”, and back-end, legacy applications running on mainframes or other legacy systems. The JI Integration runtime server environment sits in the “middle”, controlling and monitoring connections between the front-end clients and the legacy applications. A robust service API allows services to implement modern business logic to facilitate the client to host communications.

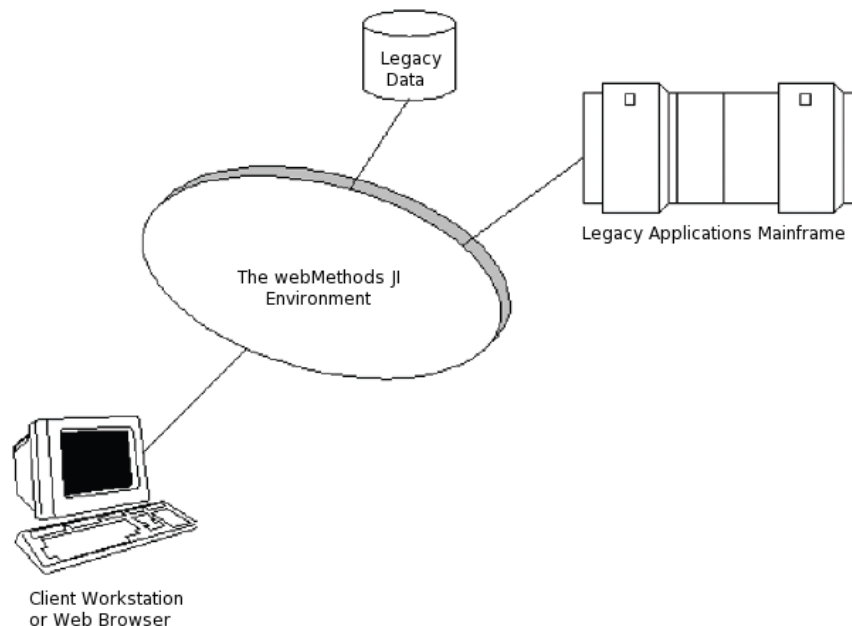


Figure 1. JI Integration is a middleware solution

JI Integration clients do not need to know the communication protocol required to communicate with the legacy host, or the location of the host application. The clients only need to know:

- The location of the JI Integration Environment Manager, a server-side process that manages all components of JI Integration environments.
- The name of the service that they need to access.

For more information about the Environment Manager, JI Integration environments, and JI Integration services, see the *JI Integration User's Guide*.

Host Communications

The following protocols are supported for communication between JI Integration services (JServices) and legacy host machines:

- TN3270/TN3270E Models 2-5: Models 3, 4, and 5 are supported with new Java services only.
- TN5250 Standard (80 column and 132 column): The session capability of TN5250E is also supported, although no other aspects of TN5250E are implemented.
- Telnet (VT100, VT220, VT320 and ANSI emulation): Telnet is used for Character Mode and referred to as such throughout this manual.

Client Libraries and Interfaces

JI Integration includes number of client libraries and interfaces that allow a variety of clients to be developed. These clients can be implemented in Java, HTML, or using a variety of interfaces to third-party tools and integrated development environments.

About JI Integration Clients

The first tier of the JI Integration solution is the client tier. JI Integration clients can be developed using the JI Integration client libraries, or clients can be implemented using interfaces to third-party tools and environments. These interfaces were created by JI Integration using the client libraries.

Client/Service Interaction: An Overview

Generally speaking, clients use JI Integration functions to send messages to the JI Integration environment. These messages provide the JI Integration environment with information about the location of the Environment Manager, the name of the service, and any input parameters that the service will use to interact with the legacy application. The steps for this client/server interaction are as follows:

- 1 The client sends a message to the Environment Manager, making a “Request for Service” (RFS).
- 2 The Environment Manager uses load balancing criteria and other information to determine which JCluster should process the client request. The JCluster then accesses or creates an instance of the JService that serves as an interface to the JI Integration service requested by the client. The client is then able to communicate directly with the service. For more information about JClusters and JServices, see the *JI Integration User's Guide*.
- 3 Once the client and service are connected, the client sends a “service message” to the service. This message consists of name/value pairs defining fields and values for those fields, for example, `acct=135`.
- 4 The service queries the legacy application based on the information provided by the client.
- 5 The service then retrieves information from the legacy application and constructs a new service message containing the new information. This message could contain data from a number of legacy application screens, or even more than one application and/or more than one legacy host.
- 6 The service then sends the new service message to the client.

Service Sessions

JI Integration clients can create “sessions” that tie multiple client requests to the same service session. This allows the client to re-connect to the same service instance that was initialized when the client first connected. The session is requested by the client using the session name; if there is no instance of the requested service that is using the session name, a new instance of the service will be created and will be tied to that session name. If there already is an instance of the requested service using the session name, the client will be connected to that existing service, assuming there is no client already connected to it.

Note: Multiple sessions with the same session name, but using different services, are allowed. However, multiple sessions using the same name and the same service are not allowed.

Sessions are created by JClient3 clients, using the `setSessionName()` method in the `ServiceConnection` class.

JI Integration Client Libraries and Interfaces

The following client libraries and interfaces are available with JI Integration:

- **The Java Client Library Version 3 (JClient3):** JClient3 is a simplified client library written in Java. It is a set of Java packages that provide the functions required for clients developed in the Java programming language to interact with JI Integration services. The JClient3 is available on all supported platforms and requires JDK 1.4.2_05 or newer.
- **Siebel Integration:** JI Integration can be integrated with the Siebel eBusiness Applications, allowing them to interact with JI Integration services. For Siebel 7 and beyond, this is accomplished through the use of the JI Integration HTTP/XML Gateway.

SOCKS Functionality

SOCKS Overview

SOCKS is a proxy protocol that allows TCP/IP clients on one side of a SOCKS server to gain access to hosts and/or applications on the other side that are not directly reachable, usually because of a firewall. After authenticating the client, SOCKS redirects connection requests to the desired host and application.

Integration Architecture

The following diagram illustrates the SOCKS configuration as it applies to JI Integration.

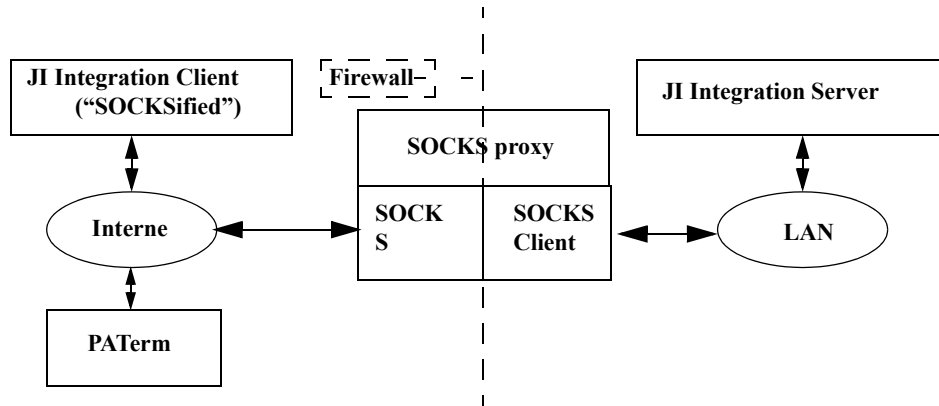


Figure 2. Integration Architecture

Currency

JI Integration maintains information about the current field, which allows you to access repeating fields and fields in repeating areas using relative access. A repeating field or area is represented as an array: `repeating_field[i]`, `repeating_area[i].field1`, or `repeating_area[i].repeating_field[j]`. This allows specific fields or areas in the array to be accessed by setting currency to the appropriate field or area using the `EA_ScreenSetCurrent()` function in your service.

For example:

```
EA_ScreenSetCurrent("Main>Screenname", "area[5]");
```

sets the current area to the fifth instance of the repeating area.

You can also access repeating fields and areas in a relative manner, using currency indicators in place of the specific subscript. The following currency indicators are allowed:

- [N]ext
- [P]revious
- [F]irst
- [L]ast
- [C]urrent

The following example code uses currency to access data in a repeating area:

```
char temp1[15];
char temp2[15];
```

```
while EA_ScreenUnload("Main>Screenname", "area[N].feature=%s
                        area[C].price=%s", temp1, temp2) == EA_OK;
{
    EA_SvcMsgLoad("servicemsg", "area[N].feature=%s,
                  area[C].price=%s", temp1, temp2);
}
```

The `EA_ScreenUnload()` function sets the current area to the next area (`area[N]`), retrieves data from the `feature` field of this area, then retrieves data from the `price` field of the current area (`area[C].price`). The code then loops, accessing data in each subsequent area in the repeating area.

Client XML Support

XML Support Overview

With the release of JI Integration version 3.5, all client libraries can send and receive XML-formatted data to/from JI Integration services.

Note: XML support is **not** provided for C services. The CGI Gateway does not support service output in XML format.

String Size Restrictions

Using Java-based JI Integration libraries within Java 1.4.2 or newer, plus Java 1.4.2 or newer for JI Integration server software will support very long strings up to $2^{64}-1$ bytes.

Client Input Message - Specifying XML Parsing

XML parsing is implemented on the server side. To specify that the XML data should be parsed, the client **must** use the predefined field names `"EA_XML"` or `"EA_XML_URI"` in the input message. When `"EA_XML"` is in the input message, the value for this field is the XML data in String format. When `"EA_XML_URI"` is in the input message, the value for this field is a reference to the XML document (URI: Uniform Resource Identifier).

When multiple `EA_XML/EA_XML_URI` fields are passed in the input message, the input map should be built as follows (the following example shows three `EA_XML` fields in input message):

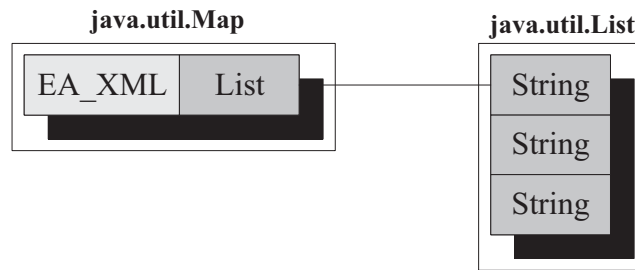


Figure 3. Three fields in input message

You may also pass the XML data “as is” to the host application. To force the XML data to be passed on to the host application un-parsed, do **NOT** use “EA_XML” or “EA_XML_URI” as the field name for the String field containing the XML data. This allows service developers to write custom code to parse the XML data.

Errors Encountered During Parsing

If the XML parser encounters any error while parsing the input XML data contained in the EA_XML field, the error will be logged, the parsing of the EA_XML field will stop, and the service front-end will continue with the next field in the input data. Thus, if an error occurs while parsing XML data, no input parameters will be set in the map for that EA_XML field.

MapMaker Generated DTD

During the client generation process, MapMaker creates a DTD describing method inputs. A stand alone DTD file is always generated.

An XML file (one for each method of each service) is only generated if “Include Sample Input Data” is checked on the Generate Code dialog (Client tab). The DTD contained within the XML file is an exact copy of the stand alone DTD file, and resides in the same directory as the client code. The XML file name is “ServiceName_MethodName.xml”. The client may then use this skeleton XML file and DTD, to create properly formatted XML data to send to the service in an input message. This ensures that the XML will work correctly with the MapMaker-generated map.

To ensure that the XML can be parsed into a valid Map/List structure and work correctly with a MapMaker-generated map, the user’s XML must conform to the MapMaker generated DTD. To achieve this, the input XML can either:

- Reference the MapMaker-generated DTD (as described above) in the DOCTYPE statement.

- Reference a DTD that is a superset/subset of the MapMaker-generated DTD in the `DOCTYPE` statement.
- Conform to the MapMaker-generated DTD (if no `DOCTYPE` specified).

If the user's XML data does not conform to the JI Integration supplied DTD, or if it contains no DTD at all, then the XML will be parsed according to guidelines outlined in *Parsing Rules*.

Parsing Rules

Overview

The following rules govern how the XML is parsed into a map structure containing name/value pairs:

- Every XML element (i.e. "name=value") will result in an entry in a List that is contained within a Map.
- Every repeating XML element at the same level will be contained in a list.
- All elements at the 1st level below the root XML level will be used as inputs to the map.

If formatting elements are used inside a `#TEXT` entry and the format needs to be preserved, then the `#TEXT` should be enclosed within a `CDATA` structure, as follows:

```
<Foo> TEXT1 [!CDATA <EM> B </EM> ] TEXT2 </Foo>
```

Otherwise, `#TEXT` elements will be parsed into `#TEXT` and child elements. White space will not be preserved unless a `CDATA` format is used.

Note: XML processing instructions contained within the input XML (e.g. "`<?InstructionText?>`") will be ignored by the parser.

Configurable Parsing Options

Four configurable properties are available to govern how the XML parser operates while parsing XML input messages. These properties appear in the properties table whenever a Service Master node is selected in the tree. They are editable for Java services only. See the *JI Integration User's Guide - Chapter 5 Configuration Manager* for information on *Setting Properties for Service Masters*.

The properties are:

XMLAttrParsingOptions - Attribute parsing option determines how XML attributes are handled by the parser.

XMLUseValidatingParser - The validating vs. non-validating parser option specifies whether the XML parser should act as a validating or non-validating parser.

XMLNamespaceAware - This option specifies whether the XML parser should be namespace aware while parsing. The XML namespace mechanism extends the XML data model to allow element type names and attribute names to be qualified with a Uniform Resource Identifier (URI). Thus, a document that describes title of a person can use title qualified by one URI, and a document that describes title of books can use title qualified by another URI. If this option is disabled, then duplicate element type names or attribute names in an XML document will result in a potential conflict.

XMLOnOutput - If there is no EA_XML field in the output map from the service, then the XMLOnOutput property is used to determine if the output should be in XML format. When enabled, the entire output map is converted into XML and sent back in one field, called EA_XML. When disabled, the entire map is sent back to the client unchanged.

Attribute Parsing Options

The XML parser supports three options as to how to handle XML attributes, by making use of the JService property XMLAttrParsingOptions. The three options for parsing attributes are:

- **Ignore all attributes in the XML input message** - Any attributes encountered while parsing the XML input message will be ignored and not included in the resulting Map/List.
- **Parse attributes into fields in the resulting Map** - Attributes encountered while parsing will be included as fields in the resulting Map/List. These attributes will be treated as sub-elements of the XML tag to which they belong.

For example, the following XML:

```
<Person Gender="male">
  <Name> Joe Smith </Name>
</Person>
```

Treating the "Gender" attribute as a sub-element of the Person tag, this XML would then be parsed as if it were written as follows:

```
<Person>
  <Gender> male </Gender>
  <Name> Joe Smith </Name>
</Person>
```

- **Parse attributes into fields in a separate Map** - Attributes encountered while parsing will be included as fields in a separate Map. The name of this map will be "EA_XML_ATTR". The service developer may choose to write custom code to access this attribute data.

As the XML parser traverses through the input XML data and generates the Map/List for the service method, it will use the “attribute parsing options” property to determine how to handle XML attributes, and then generate the resulting Map/List structure accordingly.

Examples

The following examples show what the resulting Map will look like for each of the three attribute parsing options, when the same XML file is parsed. Here is the XML file to be parsed:

```
<A>
  <Person Gender="male">
    <Name Last="Smith"> Joe </Name>
    <Addr> 123 4th Street </Addr>
  </Person>
</A>
```

This is what the resulting Map looks like when attributes are ignored. Notice that the attributes “Gender” and “Last” are not included in the Map.

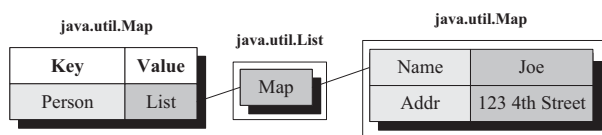


Figure 4. When attributes are ignored

The following diagram shows how the resulting Map looks when attributes are parsed into fields to be included in the Map. Here, the attributes are treated as sub-elements of the tag to which they belong. The “Gender” attribute becomes a sub-element of the Person tag, and is a field in the Map along with the Name and Addr fields (also sub-elements of the Person tag).

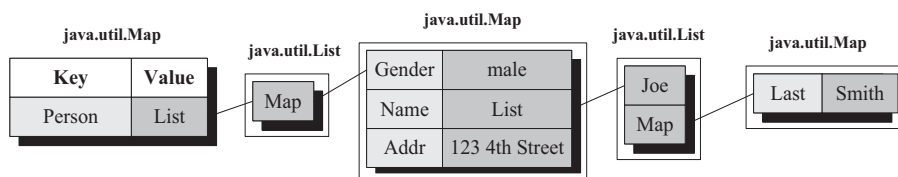


Figure 5. When attributes are parsed into fields to be included in the Map

The following diagram shows the map resulting when attributes are parsed into fields in a separate Map. Here, the attributes are included in a separate Map called EA_XML_ATTR.

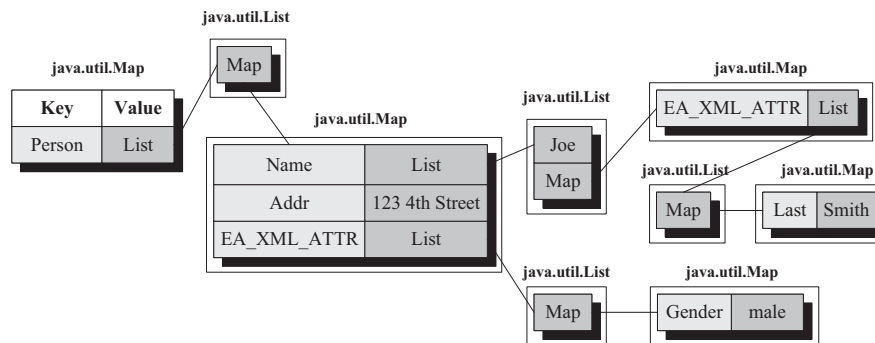


Figure 6. When attributes are parsed into fields in a separate Map

Returning XML Output to the Client

The configurable JService property `XMLOnOutput` specifies whether the output data from the service should be in XML-format. When this property is set, the reply message will contain a String field named “EA_XML” which will contain XML-formatted data.

The output XML data is created on the server side by traversing the service’s reply Map/List structure, and creating XML elements for each element in the Map/List. This XML does not conform to any specific DTD. It is simply well-formed XML defining the name/value pairs contained within the reply Map/List.

You may choose to return the XML data to the client in conformance with a specific DTD, using custom code in the service. The `EA_XML` String field can be created in custom code to contain the output XML data. When the service sends the reply message back to the client, the low-level XML-parsing service code will recognize the `EA_XML` field in the output map and assume it already contains XML-formatted data. In this case, it will simply send it, along with any other fields in the output map, straight through to the client as is.

Note: Anytime the `EA_XML` field exists in the output map from the service, the low-level parsing code will send the entire output map to the client unchanged, regardless of how the `XMLOnOutput` property is set.

If there is no `EA_XML` field in the output map from the service, the `XMLOnOutput` property is used to determine if the output should be in XML format. When this property is enabled, then the entire output map is converted into XML and sent back in one field, called `EA_XML`. If it is disabled, then the entire map is sent back to the client unchanged.

JavaServer Pages (JSP) Support

The Web Server in use must include support for JavaServer Pages. When a request arrives at the Web Server from a Web Browser, the MapMaker generated JSP is invoked. This acts as a JI Integration client and forwards the request to the appropriate service. Results are sent back to the requesting JSP, and ultimately to the Web Browser.

MapMaker creates JavaServer Pages that support the JSP standard during the client generation process. This JSP contains HTML that prompts the end user for the input data required by the service. The JSP invokes the JClient3 library (using the JavaBean interface) to invoke the service and retrieve the host output that corresponds to the specified inputs. Finally, the JSP contains the HTML to deliver the output from the host to the end user. See the *JClient API Reference - JClient3.Class* for information on settings for HTML output.

During the client generation process, two JSP source files are created: one for the standard JSP that acts as a JI Integration client, and another to handle exceptions. JavaServer Pages generated by MapMaker must be manually ported to the Web Server. See the documentation for your Web server for information on JavaServer Pages.

Java code, automatically generated during the client generation process, appears similar to the following JSP code example. Java code is contained within the marker tags `<%` and `%>`. The example code performs the following functions:

- Identifies the import packages

```
<%@ page isThreadSafe="false" import="java.util.*,
com.jacada.ea.jclient3.JClient3" errorPage="error.jsp" %>
```

- Establishes the bean ID

```
<%-- ***** --%>
<%-- get a bean instance --%>
<%-- ***** --%>
<jsp:useBean id="jclient3bean" class="com.jacada.ea.jclient3.JClient3"
scope="request" />
```

- Sets the properties for the Environment Manager, the service, and the method

```
<%-- ***** --%>
<%-- set properties of the bean --%>
<%-- ***** --%>
<jsp:setProperty name="jclient3bean" property="environmentManagerList"
value="localhost:30001" />
<jsp:setProperty name="jclient3bean" property="serviceName"
value="Service_0" />
<jsp:setProperty name="jclient3bean" property="methodName"
value="Method_0" />
<jsp:setProperty name="jclient3bean" property="*" />
```

- This HTML code creates the form presented to the user

```
<HTML>
<!-- ***** --
>
<!-- CAUTION: This file is automatically generated -->
<!-- ***** --
>

<H1><CENTER>
JI Integration<BR>
Service_0:Method_0
</CENTER></H1>
<B><CENTER>POST request - Results using JSP template</CENTER></B>
<HR>
```

```
<!-- Put a form on the browser.  It's action will be to POST the input
      and retrieve the results using the jclient3 Java bean.  -->

<FORM METHOD="POST">

<!-- Add a query string, and a text box to the form that will
      allow the user to input data for the service.  -->
<CENTER>

    <B>Keyword:</B>
    <INPUT TYPE="hidden" NAME="inputNames" VALUE="Keyword">
<BR>
    <INPUT TYPE="text" NAME="inputValues" VALUE="">
<BR>
    </CENTER>
<!-- Add the button that kicks off the action.  -->
    <BR>
    <CENTER>
    <INPUT TYPE="submit">
    </CENTER>
    <HR>
<!-- JSP jclient3 code to handle the request -->
</FORM>
```

- This evaluates whether or not there is content

```
<%
    if (request.getContentLength() > 0)
    {
%>
```

- This directs the output to HTML using a JClient3 call. Note that the tags `<jsp` and `/>` identify the actions contained within as Java code.

```
    <jsp:getProperty name="jclient3bean" property="HTMLOutput" />

<%
    }
%>
</HTML>
```

Active Server Pages (ASP.NET) Support

You must use a Web Server that includes .NET support, such as Microsoft IIS. When a request arrives at the Web Server from a Web Browser, the MapMaker generated ASP.NET client is invoked. This acts as a web service client and forwards the request to the appropriate web service, which forwards the request to the JI service. Results are sent back to the web service, which forwards the results to the requesting ASP.NET client and ultimately to the Web Browser.

MapMaker creates ASP.NET clients supporting the ASP.NET standard during the client generation process. The ASP.NET clients contain HTML and C# code that prompts the end user for the input data required by the service. The ASP.NET client invokes a service-specific library, *Jl<ServiceName>Proxy.dll*, to invoke the service and retrieve the host output that corresponds to the specified inputs. Finally, the ASP.NET client contains the HTML and C# code to deliver the output from the host to the end user.

During the client generation process one .aspx file is created for each method in the service, and one .dll file is created per service. These files are located in the *clients* directory of the path specified at the time of generation. The .dll file, *Jl<ServiceName>Proxy.dll*, is located in the *clients/bin* directory of the same path. ASP.NET clients generated by MapMaker must be manually ported to the Web Server. See your Web Server documentation for information about .NET support.

For more information about MapMaker's generation of ASP.NET sample clients, see the *Jl Integration User's Guide*.

Chapter 2. The Java Client Library Version 3 (JClient3)

This chapter describes how to use the Java Client Library version 3 (JClient3). The JClient3 provides a simplified set of Java packages that allow the development of Java-based clients to access JI Integration environments and services.

Supported Java Development Kit (JDK)

See the “Java Software Requirements” section for “JI Development Environments” in the current version of the *JI Integration Release Notes* for the minimum JDK versions required for developing JClient3 clients.

Using the JClient3

The JClient3 is an JI Integration library designed to develop Java applications that function as clients to JI Integration services. To access the JClient3 classes, import the JClient3 package into your Java client code:

```
import com.jacada.ea.jclient3.*;
import com.jacada.ea.jclient3.utility.*;
```

You should also put the jclient3.jar in your CLASSPATH when compiling and running the client.

JavaDoc

JavaDoc, in HTML form, for the JClient3 library is included with your JI Integration installation. The JavaDoc, which can be opened in any Web browser, is located in <JI_install_dir>/html/javadoc.html.

Deploying Clients Developed with the JClient3 Library

Clients developed with the JClient3 library should have access to the following items:

- jclient3.jar. The JAR file that contains the JClient3 class library. This file should be in the CLASSPATH when the client is executed.
- Paterm files. If a client-side Protocol Agent terminal window is used, the client must have access to the paterm executables and the paterm.kbm keyboard mapping file. For more information, see “SOCKS Functionality” on page 29.

Note: “SOCKSified” JClient3 clients can only use Protocol Agent terminal windows when running under UNIX.

Package `com.jacada.ea.jclient3`

The `com.jacada.ea.jclient3` package contains classes for creating connections to the JI Integration environments and services. The following sections highlight the most important classes and methods included in this package. Example code at the end of this chapter provides examples for using these and other classes included in the package (see “Example Code” on page 35).

Package `com.jacada.ea.jclient3.utility`

The `com.jacada.ea.jclient3.utility` packages provides utility classes for debugging, versioning, local host name retrieval, and text manipulation.

Creating a Connection to an Environment Manager

A connection to an Environment Manager is created using the `EnvironmentManagerConnection` class. The `EnvironmentManagerConnectionProperties` class is used to create a list of Environment Managers for use by the `EnvironmentManagerConnect` object, and to set properties related to each Environment Manager.

EnvironmentManagerConnectionProperties Class

The first step to create a connection to an Environment Manager using the JClient3 library is to create an `EnvironmentManagerConnectionProperties` object. In the following example, the host and port of the Environment Manager are passed in as parameters when the `EnvironmentManagerConnectionProperties` object is created:

```
props = new EnvironmentManagerConnectionProperties(eaHost, eaPort);
```

Where `eaHost` is the name or IP address of the Environment Manager, and `eaPort` is the port number of the Environment Manager’s server port.

`addEnvironmentManager()` Method

Next, additional Environment Managers can be added to the `EnvironmentManagerConnectionProperties` object using the `addEnvironmentManager()` method:

```
props.addEnvironmentManager(eaHost1, eaPort1);  
props.addEnvironmentManager(eaHost2, eaPort2);  
props.addEnvironmentManager(eaHostn, eaPortn);
```

Using the `addEnvironmentManager()` method to add multiple Environment Managers to the `EnvironmentManagerConnectionProperties` object allows the client to take advantage of Environment Manager redundancy in your JI Integration server environment. When the JClient3 client requests a connection to a service, it contacts the first Environment Manager listed in the `EnvironmentManagerConnectionProperties` object. If that Environment Manager cannot serve the client's request, the next Environment Manager in the list is contacted until the request is served or all Environment Managers have responded without being able to serve the request.

EnvironmentManagerConnection Class

The `EnvironmentManagerConnection` class is used to create an `EnvironmentManagerConnection` object:

```
EnvironmentManagerConnection emc =  
    new EnvironmentManagerConnection(props);
```

After the `EnvironmentManagerConnection` object has been created, a service connection is requested by using the `getServiceConnection()` method:

```
sc = emc.getServiceConnection(serviceName);
```

Where `serviceName` is the name of a service configured in your JI Integration environment.

Invoking a Method and sending Input Data to the Service

Next, to invoke the method and send a service message to the service, the service connection must be opened using the `open()` method from the `ServiceConnection` interface:

```
sc.open();
```

After the service connection is open, the `invokeMethod()` method from the `ServiceConnection` interface is used to invoke a method in the service and to pass input data, in the form of a `java.util.Map`, to the method. For information about the data types that can be included in the input Map that is sent to the service, see "Supported DataTypes" on page 26.

```
Map in = buildMap();  
Map out = sc.invokeMethod(eaMethod, in);
```

Where `buildMap()` is a user-written method that creates the input map, and `eaMethod` is the name of a method in the service.

The service message returned from the service is the `java.util.Map` object `out`.

Closing the Service Connection

To close the service connection, use the `close()` method from the `ServiceConnection` interface:

```
sc.close()
```

Using Threads with the JClient3 Library

JClient3 objects are not synchronized; as a result, if a JClient3 object is going to be shared among multiple threads, the developer must correctly handle any required synchronization.

Supported DataTypes

Input data is sent from the JClient3 client to the JI Integration service in the form of a `java.util.Map`. The Map consists of keys, representing fieldnames and names of other information that will be sent to the service, and the values that correspond to the fieldnames.

java.util.Map Supported JClient3 DataTypes	
Key	Value
String	Integer
String	Float
String	Long
String	String
String	List
String	Map

Figure 7. Supported DataTypes

Datatypes allowed as values in the key include:

- `java.lang.Integer`
- `java.lang.Float`
- `java.lang.Long`
- `java.lang.String`
- `java.util.List`
- `java.util.Map`

The java.util.List Datatype

Repeating Fields

One use for the List datatype that is allowed in JClient3 java.util.Maps is for repeating fields, the key represents the name of the repeating field and the value is a list of multiple values for the repeating field.

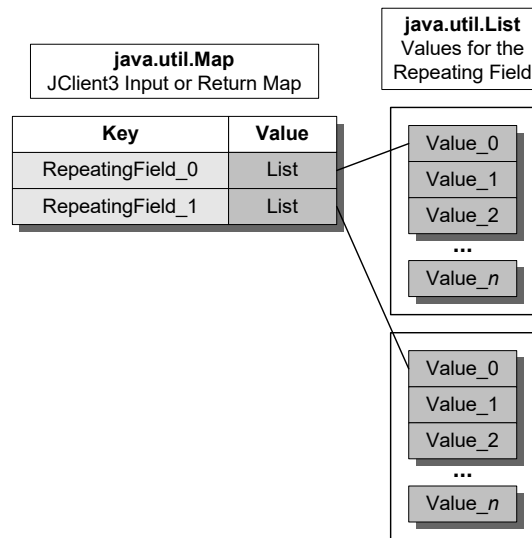


Figure 8. Repeating Fields

Table Templates/Repeating Areas

Another use for the List datatype is for a Table Template in a JI Integration Java service generated in MapMaker.

- **Table Templates:** For Table Templates (used with JI Integration Java services), the key in the Map represents the name of the Table Template, and the value represents a List which in turn contains multiple maps, one for each row in the Table Template.
- **Repeating Areas:** For Repeating Areas, the key in the Map represents the name of the Repeating Area, and the value represents a List which in turn contains multiple maps, one for each instance of the Area.

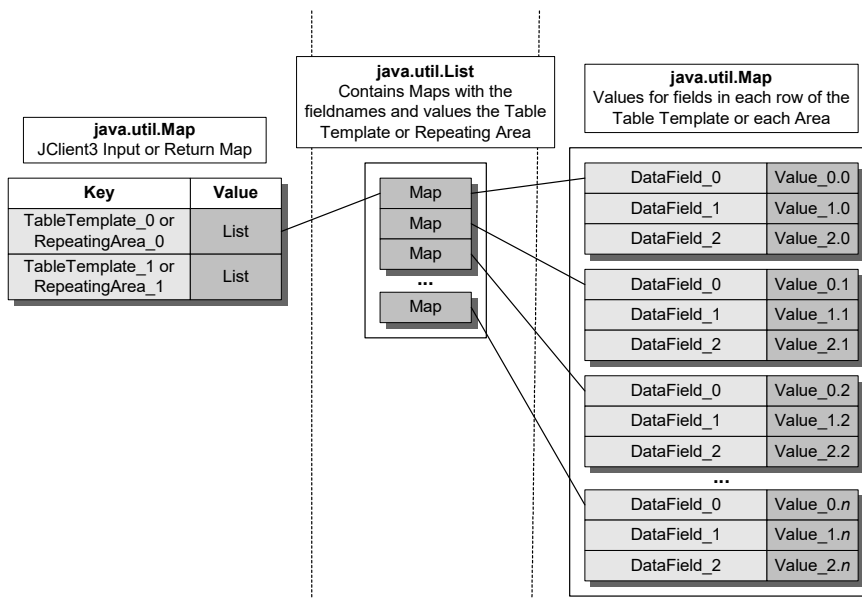


Figure 9. Table templates/repeating areas

The java.util.Map Datatype

The Map datatype that is allowed in JClient3 Maps is used for Data Templates in a JI Integration Java service generated in MapMaker. The key represents the name of the Data Template, and the value represents a Map consisting of keys that are the names of fields in the Data Template, and values that are the fields' values.

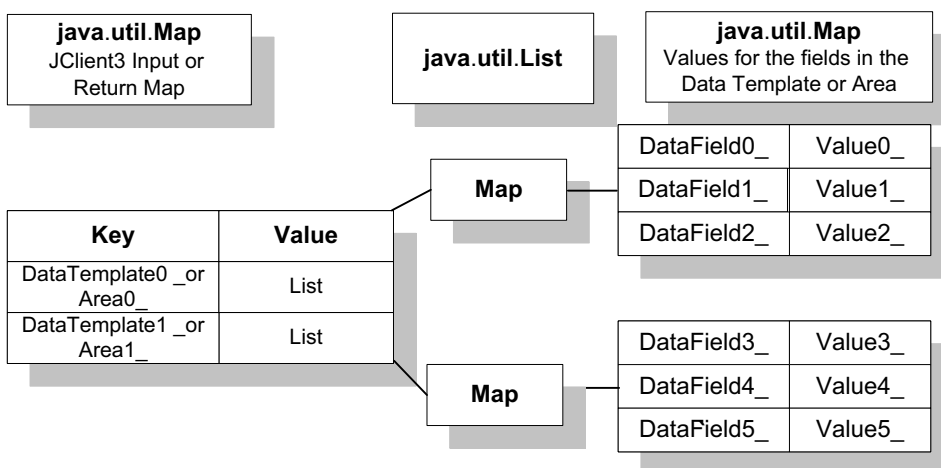


Figure 10. The java.util.Map Datatype

Note: With JI Integration Java services, Data Templates are used to extract data from the service, but are not used for input to the service. Therefore, for clients that communicate with Java services, this datatype only applies to maps returned from the service, and cannot be used as input unless the Java service is customized using Custom Classes to handle the data.

SOCKS Functionality

Java Client Changes

SOCKS V4 support is included in Java. When running a Java client application, setting the following tells the JVM to use the SOCKS proxy server:

```
java -DsocksProxyHost=host_name  
-DsocksProxyPort=port_num <client_name>
```

Java applications can also be forced to use a SOCKS proxy server by setting the following system properties within the Java source code:

```
System.getProperties().put("socksProxyHost", host_name);  
System.getProperties().put("socksProxyPort", port_num);
```

where `host_name` is the name of the SOCKS proxy server host, and `port_num` is the port number the SOCKS server is listening on for new client connections (usually 1080).

SOCKS Configuration File

To make use of SOCKS, and also use the client-side Protocol Agent Terminal Window described in the next section, the SOCKS configuration file must be set up.

JI Integration “SOCKSified” JClient3 client programs get their configuration from */etc/socks.conf*. This includes patterns run from a SOCKSified jclient3 client. It is important, that in a “SOCKSified” jclient3 application using terminal windows, the values used for proxy `socksProxyHost` and `socksProxyPort` match those in the */etc/socks.conf* file.

The format for */etc/socks.conf* is as follows:

```
deny      [*=userlist] dst_addr dst_mask [op dst_port] [: shell_cmd]  
direct    [*=userlist] dst_addr dst_mask [op dst_port] [: shell_cmd]  
sockd     [@=serverlist] [*=userlist] dst_addr dst_mask [op dst_port] [:  
shell_cmd]
```

A `deny` line is used to tell SOCKS clients when to reject a request. A `direct` line is used to tell SOCKS clients when to use a direct connection, instead of connecting through a SOCKS proxy. A `sockd` line indicates when to use a SOCKS proxy, and optionally which proxy server or servers it should try.

The simplest example would be the case where this file simply specifies the socks server which should be used for all client connections. For a SOCKS server named `xyz` the configuration file might look as follows:

```
sockd @=xyz 255.255.255.255 0.0.0.0
```

This configuration would cause “SOCKSified” clients to use the SOCKS server “xyz” for all connections regardless of the destination host.

Java plug-ins that support SOCKS v4 exist for most popular web browsers. Java applets inherit proxy settings from the browser’s proxy configuration.

Creating a Terminal Window

The JClient3 Client Library includes the ability to create a client-side terminal window so that the interaction between the JI Integration Protocol Agent and the legacy application can be displayed on the client’s machine. This is useful for debugging purposes. Additionally, a write-enabled terminal window can be created that allows the user to have live interaction with the legacy host. For a code example that creates terminal windows, see “Example 2: Protocol Agent Terminal Window Test” on page 42.

The following methods from the `ServiceConnection` interface are used to create, attach, and close terminal windows:

- `makeTerminalWindow()`
- `attachAllTerminalWindows()`
- `closeAllTerminalWindows()`

There are three methods that can be used to open terminal windows using some or all of these methods:

- 1 Call the `makeTerminalWindow()` method before the service connection is opened. Then, when the service connection is subsequently opened, the terminal window will be created and connected to the Protocol Agent.
- 2 Open the service connection, then call the `makeTerminalWindow()` method, then call the `invokeMethod()` method. The terminal window will be created and connected to the Protocol Agent when the method is invoked.

Note: This option can only be used prior to the service connecting to the protocol agent. Therefore, the service must connect to the protocol agent in the method that is invoked.

- 3 Open the service connection, then call the `attachAllTerminals()` method. The `attachAllTerminals()` method will create terminal windows and connect to all Protocol Agents currently connected to the service that don't already have terminal windows.

In each of these methods, the `closeAllTerminalWindows()` method can be used to close the terminal windows, or the terminal windows will close when the client exits.

Information Required to use Terminal Windows

Certain system information, such as the location of the `paterm` executable, is required by the `makeTerminalWindows()` and `attachAllTerminalWindows()` methods. This information is provided by using the `TerminalWindowProperties` class to create a `TerminalWindowProperties` object:

```
TerminalWindowProperties termProps = new TerminalWindowProperties();
```

The `TerminalWindowProperties` object requires the following information:

- **JI Integration Installation Directory.** The `setEAInstallDir()` method from the `TerminalWindowProperties` class is used to set the installation directory of JI Integration:

```
termProps.setEaInstallDir("C:\\JI");
```

This installation must be available in order for the JClient3 to find the `paterm` executable and then `paterm.kbm` file. These files are located in `<JI_install_dir>/lib/ProtoAgents`. The entire installation of JI Integration does not need to be present, but these two files must be accessible to the client in order to create client-side terminal windows, and must be located in the `lib/ProtoAgents` directory of directory identified in the `setEAInstallDir()` method.
- **Windows Installation Directory. Windows only.** The `setWindowsInstallDir()` method from the `TerminalWindowProperties` class sets the installation directory for Windows systems. Generally, this is `C:\\Windows` for Windows. For UNIX, this method does not need to be called.

```
termProps.setWindowsInstallDir("C:\\Windows");
```
- **Display. Unix only.** The `setXDisplay()` method from the `TerminalWindowProperties` class sets the display for UNIX installations to determine the IP address of the host on which the terminal window should be displayed. For Windows, this method does not need to be called.

```
termProps.setXDisplay("255.255.255.255:0.0");
```

After creating the `TerminalWindowProperties` object and setting its properties, the `setTerminalWindowProperties()` method from the `ServiceConnection` interface should be called to set the Terminal Window properties for an already existing `ServiceConnection` object:

```
sc.setTerminalWindowProperties(termProps);
```

At this point, the `makeTerminalWindow()` or `attachAllTerminalWindows()` methods can be called.

Using the `makeTerminalWindow()` Method to Create Terminal Windows

The `makeTerminalWindow()` method in the `ServiceConnection` class is used to create a Terminal Window. This method has the following two parameters:

- `title`: Sets the title that is displayed in the Title Bar of the Terminal Window.
- `options`: Sets various options, which are described below.

The `options` parameter consists of one or more options, with each option separated by one or more spaces. Parameters with values should be specified in `<param>=<value>` format. The possible options are:

Parameter	Value
<code>READ_ONLY</code>	The Terminal Window will operate in Read Only mode. The user will not be able to type on the keyboard.
<code>READ_WRITE</code>	The Terminal Window will operate in Read/Write mode. The user will be able to use the keyboard to type information which is then sent to the host. This mode must be set if any JI Integration methods are invoked which contain a User Interaction step.
<code>LOGFILE=<log file path></code>	Set the path to a log file. This will cause the Terminal Window to log information about what it is doing.
<code>LOGLEVEL=<level></code>	Set the logging level. The possible values are 0 to 3.
<code>REMOTE_ID=<id></code>	Set the Remote PATERM identifier.

Creating a Write-Enabled Terminal Window To create a write-enabled Terminal Window, set the options parameter to READ_WRITE as follows:

```
sc.makeTerminalWindow(terminalTitle, "READ_WRITE");
```

The `attachAllTerminals()` method from the `ServiceConnection` interface does not allow these parameters and therefore cannot be used to create write-enabled terminal windows. This method creates terminal windows with generic titles in the Title Bar and with read-only capabilities.

Creating a Write-Enabled Remote Terminal Window To create a write-enabled remote Terminal Window, set the options parameter as follows:

```
sc.makeTerminalWindow(terminalTitle, "READ_WRITE REMOTE_ID=<remote PATERM id>");
```

JavaBean Support

The following two new classes are added to the JClient3 library to provide JavaBean support:

- `JClient3.class` – This JavaBean class provides JavaBean capability for the JClient3 library. It contains methods to connect to the server, open a service, get and set properties, and invoke a method.
- `JClient3BeanInfo.class` – This class provides `BeanDescriptor` information for the JClient3 JavaBean class.

The `jclient3.jar` JAR file contains previous existing JClient3 client library classes, plus the two new classes described above.

Bean property functionality varies, based on the IDE and/or development environment being used.

Constraints on Implementations

Because the service name is a property of the JClient3 bean, there is only one service connection per bean. The user must write their own code to initialize the method name and input Map parameters for the `invokeMethod()` method.

Using a `PATerm`/`JTerm` is not supported with the JClient3 JavaBean.

There is only one set of the following properties:

- Initial delay
- Slope
- Max retries
- Timeout

Though the JClient3 library provides a separate set of the above properties for both the `EnvironmentManagerConnectionProperties` and `ServiceConnection` classes, the JClient3 JavaBean uses only one set for both connection objects.

Metadata

The JClient3 library allows you to query services for information about the service's metadata (the methods and parameters in the service). The following classes are used to retrieve metadata from the service:

- Class `ServiceMetadata`. Used to create a `Service Metadata` object and populate that object with information about the service's methods.
- Class `MethodMetadata`. Used to create a `Method Metadata` object and populate that object with information about the parameters in the method.
- Class `ParameterMetadata`. Used to get information about the parameters, such as whether it is an input and/or output parameter, and the datatype of the parameter.

For example code that uses the Metadata classes, see “Example Three: Metadata Query Test” on page 49.

Java Archive Files

The JClient3 package is provided in JAR (Java Archive) file format. Refer to Oracle's Java web pages for more information on the JAR file format .

To use the JClient3 JAR file when creating and running Java applications, add the JAR file name and location to the `CLASSPATH` environment variable.

UNIX Bourne Shell Example

```
CLASSPATH=$CLASSPATH:<JI_install_dir>/lib/jclient3.jar
export CLASSPATH
```

Windows Example

```
set CLASSPATH=%CLASSPATH%;c:\<JI_install_dir>\lib\jclient3.jar
```

Example Code

Example 1: Echo Test

```
import com.jacada.ea.jclient3.*;           // jclient3 classes
import com.jacada.ea.jclient3.utility.*;   // jclient3 utility classes
import com.jacada.util.OrderedMap;         // Map that maintains order of
added elements
import java.text.*;
import java.util.*;
import java.net.*;

public class echotest
{
    private static int    FIELDS = 5;           // number
of fields in test message
    private static int    FIELD_SIZE = 40;       // size
of fields in test message
    private static int    MAX_FIELD_SIZE = 1024; // maximum
size of a field
    private static String TEST_PHRASE = "THIS_IS_TEST_DATA___"; // data
to use to fill the message
    private static int    MIN_DEBUG_LEVEL = 0;   // minimum
level of debugging
    private static int    MAX_DEBUG_LEVEL = 3;   // maximum
level of debugging

    private DebugController debug;               // controls
debug output
    private int             debugLevel = MAX_DEBUG_LEVEL; // use
maximum debugging

    /**
     * Displays the usage of this application
     */
    void usage()
    {
        System.out.println("\n");
        System.out.println("echotest.class - " +
            "Invokes the EchoTest method in the ea_resedSvc service.");
        System.out.println("\nUsage:");
    }
}
```

```
        System.out.println("  java echotest host port");
        System.out.println("Where:");
        System.out.println("  host = JI Integration host name");
        System.out.println("  port = Service Manager port");
    } // end method usage

/**
 * Main method
 *
 */
public static void main (String args[])
{
    // create an object of this class
    echotest client = new echotest();

    // check that two arguments were passed in
    if (args.length != 2)
    {
        client.usage();
    }
    else
    {
        // call the test method, passing the args
        client.test(args);
    }
} // end method main

/**
 * Runs the test
 *
 * @param    args[]    argument list passed into the application
 * @return    zero if test completes without errors, non-zero otherwise
 */
int test(String args[])
{
    String testName = "echotest";    // name of this test
    String eaHost = args[0];         // first arg is environment manager
host
    int eaPort;                       // environment manager port

    // check for valid integer passed in for port number
    try
```

```
{
    eaPort = Integer.parseInt(args[1]);
}
catch (Exception e)
{
    // port was not a valid integer
    usage();
    return 1;
}

// JI Integration service to use
String serviceName = "ea_resedSvc";

// method in service to invoke
String eaMethod = "EchoTest";

// create a debug controller
if (debugLevel > 0)
{
    // log to file "echotest.debug"
    FileDebugTarget targ = new FileDebugTarget(testName+".debug");
    try
    {
        // create and open the debug file
        debug = new DebugController(targ, DebugController.DETAILED);
        debug.open();
    }
    catch(Exception e)
    {
        // detected some problem creating the debug controller
        System.out.println(testName +
            ": Debug setup failed - aborting");
        return 1;
    }

    // set the desired debug level
    debug.setLevel(debugLevel);
}

// create an environment manager properties object
EnvironmentManagerConnectionProperties props;
try
```

```
        {
            // create using the host and port supplied in the arguments
            // additional environment managers may be added
            props = new EnvironmentManagerConnectionProperties(eaHost,
eaPort);
        }
        catch (UnknownHostException uhe)
        {
            // couldn't resolve the host name
            System.out.println("Host '" + eaHost + "' is unknown.");
            return 1;
        }

        // create an environment manager connection (emc) object
        EnvironmentManagerConnection emc =
            new EnvironmentManagerConnection(props);

        // set debug controller and client name in the emc
        emc.setDebugController(debug);
        emc.setClientName(testName);

        // Try to get a service connection
        ServiceConnection sc;
        try
        {
            // get a connection to the ea_resedSvc service
            // (a standard service supplied with JI Integration)
            sc = emc.getServiceConnection(serviceName);
        }
        catch (Exception e)
        {
            // emc could not service the request
            System.out.println(testName +
                ": Failed to create a service connection " +
                "- aborting: " + e);
            return 1;
        }

        // enable debugging in the service connection
        sc.setDebugController(debug);

        // open the service connection
```



```
try
{
    sc.open();
}
catch (Exception e)
{
    // couldn't open the service
    System.out.println(testName +
        ": Failed to open service connection - " +
        "aborting: " + e);
    return 1;
}

// invoke the method
try
{
    // create a message map to send to the service
    Map in = buildMap();

    // invoke the method
    Map out = sc.invokeMethod(eaMethod, in);

    // get the elements of the return message
    for (Iterator i = out.keySet().iterator(); i.hasNext(); )
    {
        String name;
        String value; // only works if you know values will be String
objects
                                // EchoTest will echo back the Strings we sent

        name = (String)i.next();
        value = (String)out.get(name);

        System.out.println("Name=" + name + ", value=" + value);
    }
}
catch (Exception e)
{
    // detected a problem invoking the method or processing
    // the response
    System.out.println(testName +
        ": Error invoking service method " +
```

```
        eaMethod + " - aborting: " + e);
    return 1;
}

// close the service connection
try
{
    sc.close();
}
catch (Exception e)
{
    // detected a problem closing the service
    System.out.println(testName +
        ": Failed to close service " +
        "connection - aborting: "+ e);
    return 1;
}

// shutdown debugging if enabled
if (debug != null)
{
    // close the debug file
    try
    {
        debug.close();
    }
    catch (Exception e)
    {
        // detected a problem closing the debug file
        System.out.println(testName +
            ": Debug shutdown failed - aborting");
        return 1;
    }
}

// successfully completed the test
return 0;
} // end method test

// method buildMap - builds an input Map
// for the method to be invoked
/**
```

```
    * Builds an <code>OrderedMap</code> which will be used to create a
message
    * to the service.
    *
    * @return an <code>OrderedMap</code>
    */
Map buildMap()
{
    String name;
    String value;

    /* Fill the test buffer with test data */
    StringBuffer testBuffer = new StringBuffer(MAX_FIELD_SIZE);
    int n = MAX_FIELD_SIZE / TEST_PHRASE.length();
    for (int i=0; i<n; i++)
    {
        testBuffer.append(TEST_PHRASE);
    }
    String testString = testBuffer.toString();

    // Create a Map object
    Map map = new OrderedMap();

    // Load first field into the service map
    map.put("field1", "field1_value");

    // size the service map
    value = testString.substring(0, FIELD_SIZE);
    for (int j=0; j<(FIELDS - 2); j++)
    {
        Integer fieldNum = new Integer(j + 2);
        name = "field" + fieldNum.toString();
        map.put(name,value);
    }

    // load last field into the service message
    String fieldN = "field" + FIELDS;
    String fieldNValue = "field" + FIELDS + "_value";
    map.put(fieldN, fieldNValue);
}
```

```
        return(map);
    } // end method buildMap
} // end class echotest
```

Example 2: Protocol Agent Terminal Window Test

```
import com.jacada.ea.jclient3.*;
import com.jacada.ea.jclient3.utility.*;
import com.jacada.util.OrderedMap;
import java.text.*;
import java.util.*;
import java.net.*;

public class patest
{
    private static int      MIN_DEBUG_LEVEL = 0;      // minimum debug level
    private static int      MAX_DEBUG_LEVEL = 3;      // maximum debug level
    private static int      SLEEP_TIME = 3;          // sleep time
    private static String   NL = System.getProperty("line.separator"); //
newline char
    private static String   terminalTitle = "patest"; // title to display in
term window
    private static String   options = "TERM_OPTS=READ_ONLY"; // read only
terminal option
    private DebugController debug;                    // controls debug
output
    private int              debugLevel = MAX_DEBUG_LEVEL; // use max debug
level

    /**
     * Displays the usage of this application
     *
     */
    void usage()
    {
        System.out.println("\n");
        System.out.println("patest.class - " +
            "Invokes the get_acct method in the fp service using a client "
+ NL +
            "side terminal window to display host interaction.");
        System.out.println("\nUsage:");
        System.out.println("  java patest host port eahome winhome display");
    }
}
```

```
        System.out.println("Where:");
        System.out.println("  host      = JI Integration host name");
        System.out.println("  port      = Service Manager port");
        System.out.println("  eahome    = JI Integration installation
directory");
        System.out.println("  winhome   = Microsoft Windows installation
directory (\\\" for UNIX)");
        System.out.println("  display   = X window address for terminal window
(\\\" for Windows)");
    } // end method usage

/**
 * Main method
 *
 */
public static void main (String args[])
{
    // Create an object of this class
    patest client = new patest();

    // Check that 5 arguments were passed in
    if (args.length != 5)
    {
        client.usage();
    }
    else
    {
        // Call the test method, passing the args
        client.test(args);
    }
} // end method main

/**
 * Runs the test
 *
 * @param    args[]    argument list passed into the application
 * @return    zero if test completes without errors, non-zero otherwise
 */
int test(String args[])
{
    String testName = "patest";        // name of this test
    String eaHost = args[0];           // environment manager host
```

```
// set eaPort, check for valid integer passed in
int eaPort;
try
{
    eaPort = Integer.parseInt(args[1]);
}
catch (Exception e)
{
    // port was not a valid integer
    usage();
    return 1;
}

// set eaHome and winHome
String eaHome = args[2];
String winHome = args[3];
String display = args[4];

// set service and method names
String serviceName = "fp_svc";
String methodName = "get_acct";

// setup debugging
if (debugLevel > 0)
{
    // log to file "patest.debug"
    FileDebugTarget targ = new FileDebugTarget(testName+".debug");
    try
    {
        // create and open the debug file
        debug = new DebugController(targ, DebugController.DETAILED);
        debug.open();
    }
    catch(Exception e)
    {
        // detected a problem opening the debug file
        System.out.println(testName +
            ": Debug setup failed - aborting");
        return 1;
    }
}
```

```
        // set the desired debug level
        debug.setLevel(debugLevel);
    }

    // create an environment manager properties object
    EnvironmentManagerConnectionProperties props;
    try
    {
        // create using the host and port supplied in the arguments
        // additional environment managers may be added
        props = new EnvironmentManagerConnectionProperties(eaHost,
eaPort);
    }
    catch (UnknownHostException uhe)
    {
        // couldn't resolve the host name
        System.out.println("Host '" + eaHost + "' is unknown.");
        return 1;
    }

    // create an environment manager connection (emc) object
    EnvironmentManagerConnection emc =
        new EnvironmentManagerConnection(props);

    // set debug controller and client name in the emc
    emc.setDebugController(debug);
    emc.setClientName(testName);

    // get a service connection
    ServiceConnection sc;
    try
    {
        // get a connection to the fp_svc service
        // (an example service supplied with JI Integration)
        sc = emc.getServiceConnection(serviceName);
    }
    catch (Exception e)
    {
        // emc could not service the request
        System.out.println(testName +
            ": Failed to create a service connection " +
            "- aborting: " + e);
    }
}
```

```
        return 1;
    }

    // enable debugging in the service connection
    sc.setDebugController(debug);

    // create a TerminalWindowsProperties object
    TerminalWindowProperties termProps = new TerminalWindowProperties();

    // set the pertinent terminal window properties
    termProps.setEaInstallDir(eaHome);
    termProps.setWindowsInstallDir(winHome);
    termProps.setXDisplay(display);

    // set the terminal properties for the service connection
    sc.setTerminalWindowProperties(termProps);

    // make a terminal window
    try
    {
        sc.makeTerminalWindow(terminalTitle, options);
    }
    catch (Exception e)
    {
        // detected an error creating terminal window
        System.out.println(testName +
            ": Failed to create terminal window - " +
            "aborting: " + e);

        // try to shut down debugging
        try
        {
            if (debug != null)
            {
                debug.close();
            }
        }
        catch (Exception e2)
        {
            // ignore
        }
    }
}
```



```
    }
    return 1;
}

// open the service connection
try
{
    // terminal window will automatically attach
    sc.open();
}
catch (Exception e)
{
    // detected an error opening the service connection
    System.out.println(testName +
        ": Failed to open service connection - " +
        "aborting: " + e);
    return 1;
}

// invoke the method
try
{
    // create a message map to send to the service
    Map in = buildMap();

    // invoke the method 5 times
    for (int counter = 0; counter < 5; counter++)
    {
        // invoke the Method
        Map out = sc.invokeMethod(methodName, in);
    }

    // close the terminal window
    sc.closeAllWindows();

    // sleep for a period
    try
    {
        Thread.sleep(SLEEP_TIME*1000);
    }
    catch (InterruptedException ie)
    {

```

```
        // take no action
    }

    // attach terminal windows to all protocol agents in service
    sc.attachAllTerminalWindows();

    // invoke the method 5 times
    for (int counter = 0; counter < 5; counter++)
    {
        // Invoke the Method
        Map out = sc.invokeMethod(methodName, in);
    }
}
catch (Exception e)
{
    // detected an error invoking the method
    System.out.println(testName +
        ": Error invoking service method " +
        methodName + " - aborting: " + e);
    return 1;
}

// Try to close the service connection
try
{
    // closing the service will also close all
    // terminal windows for the service
    sc.close();
}
catch (Exception e)
{
    // detected an error closing the service connection
    System.out.println(testName +
        ": Failed to close service " +
        "connection - aborting: " + e);
    return 1;
}

// shutdown debugging
if (debug != null)
{
    // close the debug file
```

```
        try
        {
            debug.close();
        }
        catch (Exception e)
        {
            // detected an error closing the debug file
            System.out.println(testName +
                               ": Debug shutdown failed - aborting");
            return 1;
        }
    }

    // successfully completed the test
    return 0;

} // end method test

// method buildMap - builds an input Map
// for the method to be invoked
Map buildMap()
{
    // create a Map object
    Map map = new OrderedMap();

    // load first field into the service map
    map.put("acct", "7976000");

    return map;
} // end method buildMap

} // end class patest
```

Example Three: Metadata Query Test

```
import com.jacada.ea.jclient3.utility.*;
import com.jacada.ea.jclient3.*;
import java.text.*;
import java.util.*;
import java.net.*;
```

```
public class querytest
{
    private static int      MIN_DEBUG_LEVEL = 0;    // minimum debug level
    private static int      MAX_DEBUG_LEVEL = 3;    // maximum debug level

    private DebugController debug;                  // controls debug
    output
    private int              debugLevel = MAX_DEBUG_LEVEL; // use max debug

    /**
     * Displays the usage of this application
     */
    void usage()
    {
        System.out.println("\n");
        System.out.println("querytest.class - " +
            "Creates a service connection and queries a service");
        System.out.println("\nUsage:");
        System.out.println("  java querytest host port service");
        System.out.println("Where:");
        System.out.println("  host      = JI Integration host name");
        System.out.println("  port      = Service Manager port");
        System.out.println("  service = JI Integration " +
            "service name\n\n");
    } // end method usage

    /**
     * Main method
     */
    public static void main (String args[])
    {
        // create an object of this class
        querytest client = new querytest();

        // check that three arguments were passed in
        if (args.length != 3)
        {
            client.usage();
        }
        else
    }
```

```
    {
        // call the test method, passing the args
        client.test(args);
    }
} // end method main

/**
 * Runs the test
 *
 * @param    args[]    argument list passed into the application
 * @return    zero if test completes without errors, non-zero otherwise
 */
int test(String args[])
{
    String testName = "querytest"; // name of this test
    String eaHost = args[0];        // environment manager host

    // set eaPort, check for valid integer passed in
    int eaPort;
    try
    {
        eaPort = Integer.parseInt(args[1]);
    }
    catch (Exception e)
    {
        // port was not a valid integer
        usage();
        return 1;
    }

    // set eaService
    String eaService = args[2];

    // setup debugging
    if (debugLevel > 0)
    {
        // log to file "querytest.debug"
        FileDebugTarget targ = new FileDebugTarget(testName+".debug");
        try
        {
            // create and open the debug file
            debug = new DebugController(targ, DebugController.DETAILED);
        }
    }
}
```

```
        debug.open();
    }
    catch(Exception e)
    {
        // detected a problem opening the debug file
        System.out.println(testName +
            ": Debug setup failed - aborting");
        return 1;
    }

    // set the desired debug level
    debug.setLevel(debugLevel);
}

// create an environment manager properties object
EnvironmentManagerConnectionProperties props;

try
{
    // create using the host and port supplied in the arguments
    // additional environment managers may be added
    props = new EnvironmentManagerConnectionProperties(eaHost,
eaPort);
}
catch (UnknownHostException uhe)
{
    // couldn't resolve the host name
    System.out.println("Host '" + eaHost + "' is unknown.");
    return(1);
}

// create an environment manager connection (emc) object
EnvironmentManagerConnection emc =
    new EnvironmentManagerConnection(props);

// set debug controller and client name in the emc
emc.setDebugController(debug);
emc.setClientName(testName);

// get a service connection
ServiceConnection sc;
try
```

```
{
    // get a connection to the specified service
    sc = emc.getServiceConnection(eaService);
}
catch (Exception e)
{
    // emc could not service the request
    System.out.println("Error getting connection to service: " +
        eaService + ": " + e);
    return(1);
}

// enable debugging in the service connection
sc.setDebugController(debug);

// open the service connection
try
{
    sc.open();
}
catch (Exception e)
{
    // detected an error opening the service connection
    System.out.println("Error opening connection to service: " +
eaService +
                                ": " + e);
    return(1);
}

// get the service metadata
ServiceMetadata svcMetadata ;
try
{
    svcMetadata = sc.getServiceMetadata();
}
catch (com.jacada.ea.jclient3.NoSuchMethodException nsme)
{
    System.out.println("Service '" + eaService + "' does not
exist.");
    return(1);
}
catch (Exception e)
```

```
        {
            System.out.println("Error getting metadata for service: " +
eaService);
            return(1);
        }

        // output service metadata string representation
        System.out.println(svcMetadata.toString());

        // output method metadata
        if (svcMetadata.getMethodCount() == 0)
        {
            System.out.println("Service '" + eaService + "' has no
methods.");
        }
        else
        {
            // get an iterator to the methods in the service
            try
            {
                // iterate over all methods in the service
                for (Iterator i = svcMetadata.methodIterator(); i.hasNext();
)
                {
                    // get the method name
                    String methodName = (String)i.next();

                    // get the method metadata
                    MethodMetadata methMetadata =
svcMetadata.getMethodMetadata(methodName);

                    // output the method metadata string representation
                    System.out.println("  " + methMetadata.toString());

                    // iterate over all parameters in the method
                    for (Iterator j = methMetadata.parameterIterator();
j.hasNext(); )
                    {
                        // get the parameter name
                        String paramName = (String)j.next();

                        // get the parameter metadata
```



```
        ParameterMetadata paramMetadata =
methMetadata.getParameterMetadata(paramName);

        // output the parameter metadata string representation
        System.out.println("    " + paramMetadata.toString());

        // check if parameter type is Structure
        DataType paramDT = paramMetadata.getType();
        if (paramDT.getKind() == DataType.KIND_STRUCT)
        {
            // output structure members
            System.out.println("        members:");
            for (Iterator k = paramDT.dataMemberIterator();
k.hasNext(); )
            {
                DataMember dataMember = (DataMember)k.next();
                System.out.println("            " +
dataMember.toString());
            }
        }
    }
}
catch (Exception e)
{
    // detected an error reading metadata
    System.out.println("Error getting metadata: " + e);
}
}

// close the service connection
try
{
    sc.close();
}
catch (Exception e)
{
    // detected an error closing the service connection
    System.out.println(testName +
        ": Failed to close service " +
        "connection - aborting: "+ e);
}
```

```
        return 1;
    }

    // shutdown debugging
    if (debug != null)
    {
        // close the debug file
        try
        {
            debug.close();
        }
        catch (Exception e)
        {
            // detected an error closing the debug file
            System.out.println(testName +
                ": Debug shutdown failed - aborting");
            return 1;
        }
    }

    // successfully completed the test
    return 0;

} // end method test

} // end class querytest
```

Glossary

ACL	Access Control List
Actions	Used by JI Integration Java services to navigate from one legacy screen to the next on the legacy application. An action includes all data that was input by the user during trail recording in MapMaker, along with the AID key or Action that caused the screen transition.
Action Key	A key sequence that performs an action in the legacy application. Action keys are valid for the Telnet protocol and are similar in concept to AID keys.
Applet	A program written in the Java programming language that is accessed from a Web browser.
Application	A Java program run as a stand-alone program.
API	API - Application Programming Interface. The library of C or Java functions callable from UNIX and Windows programs. Used to develop JI Integration clients and services.
AID Key	The Attention Identifier Key (AID). A single key on the keyboard that, when pressed by the user, performs an action in the legacy application. Typical AID keys include the Enter and PF keys, although the legacy application may change their usage or use other AID keys. AID keys are valid for the TN3270 and TN5250 protocols.
Browser	A program that allows users to access information on a Web server. Also known as a Web browser.
CGI	Common Gateway Interface. A standard method for external gateway programs to interface with Web servers.

Character Encoding	The format or encoding of a language-set character. Character encodings are usually 1, 2, 3, or 4 bytes. Unicode is an example of a 2-byte character encoding. Other examples are ASCII, EBCDIC, and UTF-8.
Character Mode	Character mode describes the functionality in JI Integration that communicates with character-based applications over the Telnet protocol.
Client	<p>In JI Integration, client refers to one of two items:</p> <ul style="list-style-type: none">• A Runtime version of an application developed in Java that uses JI Integration client APIs to communicate with JI Integration services.• Also refers to a third-party software application that interfaces with JI Integration services and functions similarly to an application developed with a JI Integration client API.
Client Functions	The C or Java functions used to allow clients to connect to services, execute service methods, and input and extract data.
Content Pane	A content pane, also called a panel, is a GUI component that acts as a container for various GUI components. A content pane is basically a window that other GUI objects, such as buttons and text fields, are placed on.
Cookie	A general mechanism used by Web servers to both store and retrieve information on the client side of the connection.
Custom Classes	Classes that are used to “extend” or customize service code that was generated in MapMaker.
Data Field	A data field is an individual field on the legacy screen that is added to either a data template or a table template. Data fields are used in conjunction with output variables to extract data from the legacy screen.
Data Mapping	Refers to the mapping of data in the flow of a method. Every point in a method where data is sent to or retrieved from an external source requires data mapping. Data mapping is defined in the Data Mapping Editor.

Data Stream	The flow, or stream, of information between computer programs. Data on the data stream is represented using “character encoding” and is transferred using a mutually agreed upon protocol.
Data Template	A data template is a logical representation of non-repeating data fields on the legacy screen. Data templates are defined in MapMaker and are used in conjunction with output variables to extract data from the legacy application. Similar to table template, used to define repeating data fields.
Data Typing	Refers to the creation and definition of data types. Data types are defined and maintained in MapMaker’s Business Entity Editor.
DBCS	Double-Byte Character Set.
DLL	Dynamically Linked Library. A library of function calls used in Windows environments.
DOM	Document Object Model (aka “random access” protocol for XML) – an XML parser that converts the XML document into a collection of objects, which can then be manipulated in any way you choose.
DTD	Document Type Definition for XML – an optional part of the XML document prolog that specifies the kinds of tags that can be included in an XML document and the valid arrangement of those tags.
EAServiceBean	The interface between JI Integration Java service code and the JService that manages the service in the JI Integration server environment. The EAServiceBean can be extended or customized to change the interface if required.
ECS	Extended Character Support.

EIS	Enterprise Information System - an application providing information of critical importance to the day-to-day planning and/or operation of a business. EISs are generally run on larger platforms, such as mainframes or minicomputers. EISs provide the information infrastructure for an enterprise. Examples of EISs include enterprise resource planning systems, mainframe transaction processing systems, relational database management systems, and other legacy information systems.
Enterprise System	A system involved in an organization's critical business processes. Typically, enterprise systems are large and complex, use database management systems (DBMSs), and run on mainframes or minicomputers.
Formatted Fields	Fields on the legacy application that have special formatting characteristics. MapMaker identifies all such fields on the legacy screen and uses the field layout to match screens (unless the fields are disabled and Tags are used to identify screens).
GBBasic	A Java package, <i>com.jacada.mapstudio.GBBasic</i> , that is included with JI Integration and can be used to extend or customize Java service code that was generated in MapMaker.
GUI	Graphical User Interface. An application that allows users to interface with computer programs in a graphical environment. In JI Integration, MapMaker, the Configuration Manager, and the System Monitor are all Graphical User Interfaces.
HTML	HyperText Markup Language, a format used to create Web documents.
Hos	A computer machine where applications reside. In JI Integration, hosts can be the machine on which components of the JI Integration environment are running, the machine on which the telnet, TN3270, or TN5250 server reside, or the machine on which the legacy applications reside.
IBE	Internal Business Entity. Refers to data types that are used internally by the JI Integration Service. A global variable may be defined of an IBE data type.
IDE	Integrated Development Environment. Refers to a graphical development tool that uses standard GUI components to facilitate application development.

Jacada Integrator	See JI Integration.
Java	An object-oriented, platform-independent programming language developed by Oracle.
Java services	JI Integration services developed using the Java programming language. Java services are generated from the MapMaker graphical development interface (GUI) and can be customized using the custom service classes included with JI Integration.
JClient3	The Java Client Library version 3 is an improved version of the JClient, which allows JI Integration clients to be developed using JDK 1.4.2_05 or newer.
JDBC	Java DataBase Connectivity.
JDK	Java Development Kit. The development environment for the Java programming language. Includes a Java Runtime Environment.
JRE	Java Runtime Environment. The minimum environment required to run Java applications. This is a combination of a JVM along with the core classes and files required to run Java applications.
JVM	Java Virtual Machine. A Java interpreter that converts Java code into executable code.
Legacy data	Data residing on a mainframe platform.
Legacy host	The machine or host on which legacy applications reside.
Map	The logical representation of the screens, fields, data input and AID keys that make up the user interaction with a legacy application. Maps are created in MapMaker. Note that JI Integration's use of the term Map is distinct from the java.util.Map that is included with Java.

MapMaker	The graphical development interface provided with JI Integration for the purpose of developing Java services. Used to record trails, maps, data and table templates, create methods and services, and then generate and optionally deploy the services into the JI Integration server environment.
Methods	Object-oriented entity of one or more functions. A collection of methods, initialization code, and events make up a service.
MLM	Map-List-Map. Refers to an XBE data type used for communication with a client, such as a Java, C, or VB Client.
Multicasting	A connectionless IP networking communication in which applications on the IP network broadcast information over a well-known socket.
Multithread-safe	See thread-safe.
Multithreaded	See threaded.
Offset	Refers to the location of data on the legacy screen. The offset is determined using the following format: For an 80 column screen, the offset is (column # - 1) + 80 x (row # - 1). For example, the first column of the second row is position 80: (1 - 1) + 80 x (2-1).
Package	A collection of java classes that are grouped together to form a logical combination of classes.
Presentation Space	A representation of the communications between the legacy host to the JI Integration environment, including the screen and field information from the legacy application.
Protocol Agent	A software interface that governs the procedures used to exchange information between physically remote entities such as computer systems. The Protocol Agent governs the format of the messages, the generation of checking information, and the flow control, as well as the actions to take in the event of errors.
Proxy Server	Special instances of Resource Servers that allow multicasting communication to take place over multiple sub-nets.

Resources	The components of JI Integration that are managed by the Resource Server. These components include Resource Databases and license files.
Resource Database	A database that is used in the JI Integration environment to store environment and service configuration, along with service code and maps.
Resource Serve	rManages the communication between environment managers and the JI Integration resources.
RMI	Remote Method Invocation. A standard from Oracle that allows distributed Java objects to communicate with each other over TCP/IP networks.
Screen	The screen, as contained in the data stream, that is coming from the legacy host.
Screen Mapping	The process of marking the physical boundaries of, and defining the screen components found on, an external application screen. The components include tags, fields, areas, repeating areas, and repeating fields.
Service	A collection of methods which answer requests from a client.
SOCKS	SOCKS is a firewall proxy protocol.
System Monitor	The tool used to monitor the JI Integration System. It allows you to monitor, log, and view real-time activity for all or selected Environment Managers, JClusters and JServices, clients, and services.
Table Template	A Table Template is a logical representation of the area on a legacy screen that contains repeating Data Fields. Table templates are defined in MapMaker and are used in conjunction with output variables to extract data from the legacy application. Similar to Data Template, used to define non-repeating Data Fields.
Tag	A user-defined component of the host application screen that most often serves as a label for fields. You can define any static screen text as a tag. MapMaker can identify external application screens by the tags that are defined for them.

Telnet	A TCP/IP-based terminal emulation protocol. Requires a Telnet server. Telnet is also used to describe the Character Mode functionality within JI Integration.
Terminfo	UNIX terminal information database. See the UNIX terminfo(4) man page.
Thread-safe	Also multithread-safe (MT-safe). A description of a function or library that may be called in a threaded environment without any additional coding.
Thread	A single flow of control within a process or address space. Programs using two or more threads are referred to as threaded or multi-threaded.
Threaded	Also multithreaded. A form of multi-tasking that uses multiple independent execution threads.
TN3270	An implementation of the telnet protocol that is used to communicate between TCP/IP networks and IBM mainframe applications that use IBM 3270 terminals. A TN3270 server is required for connection from the TCP/IP network to the mainframe.
TN5250	An implementation of the telnet protocol that is used to communicate between TCP/IP networks and IBM AS400 applications that use IBM 5250 terminals. A TN5250 server is required for connection from the TCP/IP network to the AS400.
Trail	The linear path of all screens encountered during navigation through a host application. MapMaker records trails during host application interaction.
Unicode	A universal character code.
Web	A network of computers based on the client-server model. The Web uses Web browsers to access information from a Web server. A Web can be a part of the World Wide Web or can be a part of a separate network, also known as an "Intranet".
Web browser	A program that allows users to access information on a Web server.

JI Integration	Consists of one or more Environment Managers, Resource Servers, resources including the Resource Database, and JI Integration clients and services.
XBE	eXternal Business Entity. Refers to data types that are used for the JI Integration Service to send and receive data to and from an external source. Such external sources are Legacy Screens and Clients.
XML	eXtensible Markup Language – a text-based markup language that is fast becoming the standard for data interchange on the web.
XSD	XML Schema Definition. Specifies how to formally describe the elements in an XML document. One of the XBE data types supported by MapMaker is XML/XSD.

Index

A

Action Key	57
Actions	57
Active Server Pages	21
AID Key	57
API	57
Applet	57
Application Programming Interface	57
ASP.NET	21

B

Browser	57
---------------	----

C

CGI	57, 58
Character Encoding	58
Character Mode	58
Client	58
functions	58
Client XML Support	12
Cookie	58
Currency	11

D

Data Field	58
Data Mapping	58
Data Stream	59
Data Templates	59
Data Typing	59
DLL	59
Document Object Model (DOM)	59
Document Type Definition (DTD)	59

Documentation	5
Viewing on-line	6
DOM	59
DTD	59

E

EAServiceBean	59
ECS	59
Extensible Markup Language (XML)	65

F

Fields	
formatted	60
Formatted Fields	60
Formatting Conventions	4

G

GBBasic	60
GUI	60

H

Host	60
HTML	60

I

IBE	60
IDE	60
Integration architecture	11

J

Java	
runtime environment (JRE)	61
virtual machine (JVM)	61
Java Client Library	10
Java Development Kit (JDK)	61
Java Runtime Environment (JRE)	61
Java Services	61
Java Virtual Machine (JVM)	61
JavaServer Pages	18
JClient	10, 61
JDBC	61
JDK	61
JI Integration environment	61, 65
JRE	61
JVM	61

L

Legacy Data	61
Legacy Host	61

M

Map	61
MapMaker	62
Methods	62
Multicast	62
Multithreaded	62
Multithread-safe	62

O

Offset	62
--------	----

P

Package	62
---------	----

Presentation Space	62
Protocol Agent	62
Protocol Agent terminal windows	24
protocols	8
Proxy Server	62

R

Resource Database	63
Resource Manager	63
Resources	63
RMI	63

S

Screen	63
mapping	63
Service	63
Siebel Integration	10
SOCKS	29, 63
SOCKS Functionality	29
SOCKS functionality	29
SOCKS Overview	10
System Monitor	63

T

Table Template	63
Tag	63
Telnet	8, 64
Terminfo	64
Thread	64
Threaded	64
Thread-safe	64
TN3270	8, 64
TN5250	8, 64
Trail	64

U

Unicode64

V

VT100/ 220/3208

W

Web64

Web Browser64

X

XBE65

XML65

XML Support Overview12
